

Spec□Box

User Manual

©Adelard 1996. All rights reserved worldwide.

The software described in this document is furnished under a licence and may be used or copied only in accordance with the terms of such a licence.

No part of this publication may be copied or distributed, transmitted, transcribed, stored in a retrieval system, or translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, manual, or otherwise, or disclosed to third parties, without the express written permission of Adelard.

Manual version PC/2.21a: 21st May 1996

Adelard

Coborn House, 3 Coborn Road, London E3 2DA, UK

Telephone: +44 (0)181 983 1708

Fax: +44 (0)181 983 1845

All trademarks acknowledged.

CONTENTS

1	Introduction	1.1
1.1	Overview	1.1
1.2	Method of operation	1.3
2	Installation	2.1
2.1	General	2.1
2.2	Use of a mouse	2.1
2.3	PC versions	2.2
3	General features	3.1
3.1	The menu system	3.1
3.2	The help system	3.2
3.3	Displays	3.3
3.4	Aborting the current activity	3.3
3.5	Batch processing	3.4
4	Utilities	4.1
4.1	Selecting the input file	4.1
4.2	External edits	4.2
4.3	Configuration	4.3
4.4	DOS	4.5
5	The syntax checker	5.1
5.1	Syntax checker options	5.1
5.2	General operation	5.2
5.3	Selecting the start point	5.3
5.4	Selecting an insertion point	5.4
5.5	Action on error	5.4
5.6	Adding and correcting text	5.5
5.7	Editor	5.9

6	The analyser	6.1
6.1	Arities	6.1
6.2	VDM quotation expressions	6.3
6.3	Scope	6.4
6.4	Error messages	6.7
6.5	Caveats	6.13
6.6	Output during checking	6.14
6.7	Report file	6.14
6.8	Listing file	6.15
6.9	Cross-reference file	6.15
7	L ^A T _E X generator	7.1
7.1	General operation	7.1
7.2	Formatting	7.1
7.3	Comments	7.2
7.4	Line numbering	7.3
7.5	Subscripts	7.4
7.6	Running L ^A T _E X	7.5
7.7	bsivdm.sty	7.6
8	Mural translator	8.1
8.1	Syntactic variants	8.1
8.2	Unsupported classes	8.2
9	The SpecBox grammar	9.1
9.1	EBNF concrete syntax notation	9.1
9.2	Documents	9.2
9.3	Modules	9.2
9.4	Type definitions	9.4
9.5	Type expressions	9.4
9.6	State definition	9.7
9.7	Constant definitions	9.7
9.8	Function definitions	9.7

9.9	Operation definitions	9.9
9.10	Statements	9.10
9.11	Expressions	9.13
9.12	Bindings and Patterns	9.20
9.13	Identifiers and basic tokens	9.22
9.14	Index to the grammar	9.24
9.15	Keywords	9.28
9.16	Operator precedence	9.29
9.17	Lexical rules	9.30
10	Illustrative examples	10.1
10.1	General observations	10.1
10.2	Notes on the ASCII syntax	10.2
10.3	Telegram analysis	10.4
10.4	Code	10.8
10.5	Bank Accounts	10.13
10.6	Binary Trees	10.17
11	Modules	11.1
11.1	Background	11.1
11.2	Sharing	11.8
11.3	Exports and imports	11.12
11.4	Instantiation and parameterization	11.13
12	Bibliography	12.1
13	Index	13.1

1 INTRODUCTION

SpecBox enables you to check formal specifications written in draft ISO VDM for syntax errors and certain semantic errors. This User Manual gives comprehensive instructions for the operation of *SpecBox*. It also describes the VDM grammar used by *SpecBox*, giving the concrete syntax and some illustrative examples. Firstly, however, the structure of *SpecBox* and its mode of working are briefly outlined.

1.1 Overview

The structure of *SpecBox* is illustrated schematically in Figure 1. It is composed of the syntax checker, the analyser, the L^AT_EX generator and the Mural translator, plus a top level program that invokes these as required, and provides file selection, external editing and configuration.

The syntax checker reads the input file in ASCII format and analyses it to produce a *parse tree*. If the file contains a syntax error, an error message is displayed together with some diagnostic information. The built-in editor can then be used to correct the error, or an external editor can be used for major changes. A log file is produced of each parsing session.

When the input file is syntactically correct, the parse tree is examined by the analyser for various semantic errors. Diagnostic messages are displayed on the screen while this analysis is taking place, and a report file, an annotated listing and a cross-reference file are produced.

SpecBox will also generate a file of L^AT_EX macros to display

the input file using the draft ISO VDM mathematical syntax. A version of the parse tree can also be produced that can be input to the Mural [3] tool for the generation and discharge of proofs about the specification.

The system includes an on-line help facility which can be invoked by pressing **F1**. When a menu is displayed, the help is specific to the highlighted item; otherwise, it relates to the current activity.

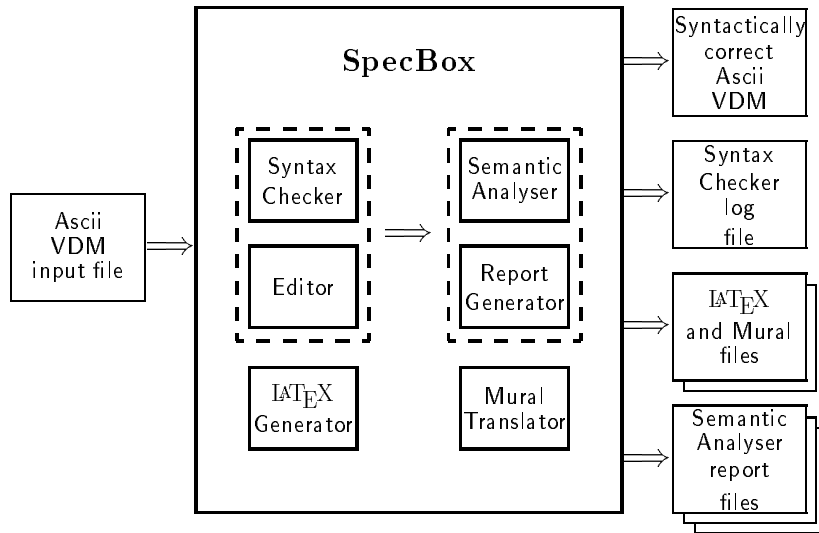


FIGURE 1: OUTLINE STRUCTURE OF *SpecBox*.

1.2 Method of operation

This subsection outlines the general way *SpecBox* is used. Refer to sections 3 to 7 for a detailed description of the functions provided.

A normal *SpecBox* session begins by typing **specbox** at the computer terminal. After the system has loaded, the top level menu is displayed with the **File** item highlighted; this is selected by pressing the **Enter ↵** key or the letter **F**. The input file is then selected from the resulting menu, or the pathname typed in.

After a file has been selected, it is read into the built-in editor, and the top level menu reappears with the **Module** item highlighted, with other syntactic units that can be analysed, such as *operation definition* and *expression*, displayed below. The analysis does not have to begin at the start of the input file: the built-in editor can be used to position the start of the analysis anywhere in the file.

Selecting the appropriate syntactic unit causes parsing to commence. The input file is listed to the screen as the analysis progresses. If parsing is successful the message **CHECK OK** is displayed, and the system returns to the top level display.

If a syntax error is encountered, a *beep* is emitted and an error banner is displayed towards the bottom of the screen, with diagnostic information below it. The listing pauses at the word or symbol that caused the problem, with the offending token highlighted. The correction menu that appears contains alternative tokens as well as a general edit option; usually the error is fairly simple to correct and the required token is

selected from the menu, which automatically replaces the highlighted one on the screen. If necessary, the structure of the grammar can be interrogated to determine which token is correct. Alternatively, the **Editor** option can be chosen if the error is such the required replacement does not appear on the menu. Whichever course is chosen, the system is put into edit mode, which allows errors to be corrected using normal word processor functions. Pressing **F8** concludes the edit session and causes the parse to recommence. Once the input file is altered, the system will ask the user whether to save the edited version before moving on to analyse the file.

If you do not wish to correct the error, you can jump past it by selecting the **Skip** option on the edit menu; this causes parsing to start again at the beginning of the next major syntactic item in the file.

Once a syntactically correct file has been produced, the top level menu will reappear with the **Analyse** item highlighted. Selecting this option causes semantic analysis to commence. Diagnostic information is displayed as this analysis proceeds; at the end, three new files are created: a report file, giving a summary of the analysis; an annotated version of the input file, indicating where semantic errors have occurred; and a cross-reference listing, indicating where each declaration is made and used.

If any semantic errors have occurred, they may be put right by selecting the **External edit** item on the top menu, before running the syntax checker and analyser again.

A syntactically correct file may be converted to the mathematical syntax by selecting the **LaTeX** option on the

main menu. This causes a file of L^AT_EX commands to be generated that will print a document in the mathematical notation. If only part of the source file has been syntactically checked, this section will be converted and the remainder will be copied unchanged.

SpecBox can also be run in batch mode by giving it a list of files to be analysed. It will check the syntax of each in turn, jumping past any errors that occur and writing the results to the log file. If there are no syntactic mistakes, *SpecBox* will go on to semantic analysis and L^AT_EX generation, and if there are no semantic errors it will perform Mural translation.

2 INSTALLATION

2.1 General

SpecBox runs under Microsoft MS-DOS 3.2 and higher.

Installing *SpecBox* merely involves making a directory called **specbox** on your hard disk and copying the diskette supplied into it.

You will probably wish to use *SpecBox* from other directories, so you should amend the **path** command in **autoexec.bat** accordingly.

Your **config.sys** file must contain the command

```
files=20
```

or you will get system errors during startup. Once you have added this line to the file, you must reboot the machine so that it takes effect. It may be necessary to increase the number of files in this command to 30 or 40 when running *SpecBox* in a multi-tasking environment, such as Microsoft Windows.

2.2 Use of a mouse

SpecBox can be used with a Microsoft compatible mouse. The

Configure

 menu item can be used to enable or disable the use of a mouse. Enabling the mouse takes effect straight away; disabling takes effect from the next time *SpecBox* is loaded. See Section 4.3 for details of how to carry this out. *SpecBox* assumes that the mouse uses interrupt 51; if this is not the

case the system may crash if the mouse is enabled.

SpecBox does not incorporate a mouse driver, and a suitable one must be installed before running *SpecBox*.

2.3 PC versions

Two PC versions of *SpecBox* are available: the standard and the 80386 versions.

80386 version

The 80386 version uses a DOS extender that is compatible with both Microsoft MS-DOS 5.0 and Microsoft Windows 3.

This version stores the concrete parse tree entirely in RAM, and its performance will therefore depend on the amount of RAM in the particular configuration. The 80386 version is normally shipped configured so that *SpecBox* occupies extended memory; this leaves the standard 640kB for other DOS applications, and in particular the external editor (see Section 4.2). This configuration may run out of heap space when analysing specifications over about 10 pages unless the PC contains at least 2MB of RAM. Configurations with less RAM than this will probably perform better with the standard (8086) version. 80386 PCs with 4MB or over of memory should be able to analyse any realistically sized specification.

SpecBox sorts the cross-reference information produced by the analyser. Unusually large and complex specifications may cause it to run out of workspace during this sorting; if this

occurs, an unsorted listing can be selected through the **Configure** menu item. See Section 4.3 for details of how to carry this out.

SpecBox makes use of a number of working files during its operation, and because the root directory can only hold a limited number of files it is desirable to place these in a subdirectory, which in the 80386 version can be specified through the **Configure** menu item. If no subdirectory is configured, the setting of the ‘TEMP’ environment variable will be used if it is defined.

Standard version

The standard version has been written for 8086 and 80286 PCs. This version is constrained to conventional memory and uses disk storage for parse trees too large to hold in RAM; this imposes certain limits on the size of specification that can be analysed. The absolute limit is that the concrete parse tree must occupy less than 1MB: the number of pages this represents obviously depends on the nature of the specification and the number of comments (comments are not retained by *SpecBox*), but is very roughly fifteen pages of uncommented VDM.

You will find, however, that *SpecBox* slows down, and the amount of disk activity increases, when the concrete parse tree becomes too large to hold in actual memory, and corresponds to about three pages of uncommented VDM. If your specifications include modules that are much larger than this, we suggest that you check them in fragments of around four or five pages while developing them, selecting the appropriate

syntactic category from the main menu, and then check the complete module at the end.

The performance of the standard version can be improved by freeing as much conventional memory as possible for *SpecBox*. Using Microsoft MS-DOS 5.0, or an alternative memory manager such as that from Quarterdeck, may enable you to move some memory-resident software into the upper memory area and, if it is available, expanded or extended memory, on PCs with 80286, 80386 or 80486 processors. If this is not possible or not sufficient, you will need to remove optional memory-resident software, such as mouse drivers, network drivers and shells. *SpecBox* briefly displays at the bottom of the screen the amount of heap available after the core system is loaded; 385 kbytes available heap is the minimum to run *SpecBox* satisfactorily, 420 kbytes is recommended, and 435 kbytes should be achievable with full use of memory management.

If you have a PC with additional memory configured as a virtual disk, the performance of the standard version can be improved on large files by configuring *SpecBox* to store the parse tree on the virtual disk. See Section 4.3 for details of how to carry this out.

3 GENERAL FEATURES

This section describes the general features of *SpecBox* that apply to all the functions, such as the menu system and the help facility. The functions themselves are covered in sections 4 to 7. Batch processing is also described.

3.1 The menu system

SpecBox is principally controlled by means of a system of menus, with some additional information entered through edit boxes.

You can select items from menus in three ways:

- (1) By pressing the appropriate *quick key*. This is the leftmost capital letter in the item name, and is the initial letter unless two or more items start with the same letter.
- (2) If the mouse is not in use, by moving the highlighting bar to the required item using the ↑, ↓, Home and End keys, then pressing Enter ↵.
- (3) With the mouse, by pointing at the item and clicking the left button.

A small arrow (↓) to one side of the menu indicates that there are more menu items than currently displayed. If the mouse is not in use, the ↑, ↓, Home and End keys will cause the menu window to scroll to display the other items. If the mouse is in use, clicking the small arrow (↓) scrolls the menu.

Not all menu items may be allowed at a particular stage in analysis: for example, \LaTeX generation cannot proceed until syntax checking has been carried out successfully. Such items are shown in brackets, and the highlighting bar will not stop on them, and their quick keys are not effective.

Help on the currently highlighted item is obtained by pressing **F1**; the help system is described more fully below.

Most menus can be cancelled by pressing **Esc** or clicking the right mouse button; the only exceptions are those where no clear default action exists.

3.2 The help system

SpecBox has a comprehensive help system that provides information and guidance at all stages of its operation. The help information is arranged as a tree, with each node containing some help text and a menu leading further down the tree; the leaves have empty menus. The menus also allow direct access to the root (symbolised by ‘\’ in the menus) and to the last help item selected.

You can invoke the help system by pressing **F1** at any time, or by selecting the **Help** item from the top level menu.

The help obtained by pressing **F1** depends upon the state of the system in the following way:

at a menu The help is specific to the highlighted item in most cases. The exception occurs at menus that ask for confirmation of a specific command or situation (e.g. *Save altered file?* or *End of text?*), where the help

explains the reasons for the question and the consequences of the alternatives.

at an edit box The help summarises the edit commands.

in the built-in editor The help summarises the edit commands and also provides access to a list of the VDM reserved keywords.

during analysis The help provides general information on the operation in progress.

Leave the help system by typing Esc or clicking the right mouse button.

3.3 Displays

SpecBox uses the bottom two lines of the display for informative messages during analysis. Generally, the input filename is displayed on the left of the upper of these, with the remainder reserved for specific error, help and status messages; these are described fully in sections 5 to 7.

A **G** in the upper right-hand corner of the display indicates that garbage collection is in progress.

3.4 Aborting the current activity

Syntax checking, semantic analysis, \LaTeX generation and Mural translation can be abandoned by pressing Esc or clicking the right mouse button. A menu headed **Quit?** will

appear; select ☐ **Yes** ☐ to return to the main menu, or ☐ **No** ☐ to continue the activity.

3.5 Batch processing

SpecBox can be run in batch mode as well as the more normal interactive mode. To carry out batch analysis, set up a file containing a list of the files to be analysed; the list should be separated by spaces, and may continue over several lines. The file names should obey the conventions described in Section 4.1.

In order to run in batch mode, *SpecBox* must be invoked using the command line:

```
specbox -b <filename>
```

where <filename> is the file containing the list of files for analysis.

When *SpecBox* is invoked, it will analyse each of the files in turn. It will firstly carry out syntactic checking, jumping past any errors that are encountered in exactly the same way as the *continue* facility. In batch mode, *SpecBox* writes a log file of the checking session, with the file extension `‘.log’`.

If there are no syntactic mistakes, *SpecBox* normally goes on to semantic analysis and L^AT_EX generation, and if there are no semantic errors it will perform Mural translation. The extent of analysis in batch mode can be restricted by using one or more of the following switches on the command line:

- s syntax check only
- a syntax check and semantic analysis only
- l syntax check and L^AT_EX generation only
- m syntax check, semantic analysis and Mural translation only

All switches can alternatively be written /**x**.

During batch processing, 'BATCH MODE' is displayed at the bottom left of the screen.

When batch processing is complete, and if all the files were successfully opened, *SpecBox* returns to the operating system. If *SpecBox* was unable to open any of the files in the list, an error message is printed containing the names of the files not analysed.

Batch mode can be aborted by pressing Esc or clicking the right mouse button, in which case *SpecBox* goes into interactive mode.

4 UTILITIES

This section describes the utilities reached from the top level menu for selecting the input file, carrying out edits in conjunction with an external editor, configuring the system, and accessing the operating system.

4.1 *Selecting the input file*

Top level menu File

This function is selected to specify the source file at the start of a *SpecBox* session, and to change the source file during the course of a session.

When selected, File produces a list of the files with the extension `.vdm` in the current directory, which can be directly selected. Alternatively, selecting the Enter item gives an edit box into which you type the file name in standard DOS notation, terminating it with a Enter ↵. If you omit the file extension, *SpecBox* assumes it to be `.vdm`. You can specify a file name without an extension by terminating the name with a `.'`. Errors can be corrected using the ←, ⇐, Del, Home and End keys.

SpecBox checks that the specified file exists and that you have write access to it, and displays an error message if that is not the case. Pressing any key removes the message and allows the filename to be entered again.

You can change the default directory by selecting

SpecBox User Manual, version: PC/2.21a

Change directory. The normal conventions of ‘.’ for the current directory and ‘..’ for the parent can be used. The current directory is displayed at the bottom of the screen during file selection.

Pressing **Esc** or clicking the right mouse button returns to the main menu without affecting any previously entered filename.

SpecBox will load a file at startup if it is invoked with the command line:

```
specbox -f <filename>
```

or

```
specbox /f <filename>
```

4.2 External edits

Top level menu **ExterNal edit**

This function enables you to correct semantic errors without leaving *SpecBox* by giving access to an external editor of your choice. External editing can also be used to put right syntax errors that cannot be corrected using *SpecBox*’s internal editor.

The path to the editor is set up using the **Configure** command described in Section 4.3.

When **ExterNal edit** is selected, the screen blanks and control is passed to the external editor, which is instructed to

load the input file to *SpecBox*. The editor is then used in the normal way. When the edit is complete and the file saved, leaving the editor causes *SpecBox* to resume operation.



Since errors may have been introduced during editing, *SpecBox* insists on carrying out syntax checking again on any file that has been externally edited, even if it was syntactically correct before.


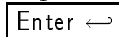
4.3 Configuration



Top level menu Configure

The configuration command allows you to select whether or not *SpecBox* will *beep* if it detects an error, to specify the path to an external editor, to select the disk to be used for storing large parse trees (8086 version) or the temporary subdirectory (80386 version), to enable use of the mouse, and to disable cross-reference sorting.




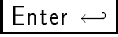
When Configure is selected, an edit box appears headed **Editor** *pathname*, with the path to any editor previously set up shown in the box. Enter the pathname of the editor in standard DOS format, and terminate the entry using the Enter ↵ key, or press Esc or Enter ↵ alone, or click the right mouse button, to use the existing path. The editor is called with this pathname followed on the same command line by the input file selected using the *SpecBox* File command. *SpecBox* does not check that the specified path exists, and will give the DOS error **Bad command or filename** when the ExterNal edit command is called if it does not.

If you wish to disable the external edit function, clear the pathname using the  key and then press .

When the external editor has been specified, another edit box appears, headed **Beep on or off?**. Enter **on** or **off** according to whether or not you want an audible warning of an error, such as an incorrect input file specification or a syntax error in the input file. Press  or  alone, or click the right mouse button, to retain the old setting. *SpecBox* will only accept the words **on** and **off** in response to this item.

Next, an edit box appears headed **Sort cross-references?**. Enter **yes** or **no** according to whether or not you want the cross-reference file produced by the analyser to sort the entries into ASCII order. Press  or  alone, or click the right mouse button, to retain the old setting. *SpecBox* will only accept the words **yes** and **no** in response to this item. See Section 6.9 for more information on the cross-reference file.

Next, an edit box labelled **Virtual file drive?** will appear. The purpose of this depends on whether you are using the 8086 or 80386 versions:

8086 version Enter the drive name (without a following ':') for the disk you wish the parse tree to be stored on if it becomes too large to store in memory; you should obtain the best performance from your system by specifying a virtual disk if your PC is configured in that way. To specify the current drive, clear the pathname using the  key and then press . Press  or  alone, or click the right mouse button, to retain the old setting.

80386 version Enter the pathname of the subdirectory in which temporary working files are to be stored. If no subdirectory is configured, the setting of the ‘TEMP’ environment variable will be used if it is defined; otherwise the root directory of the current disc will be used. Press or alone, or click the right mouse button, to retain the old setting.

You can correct errors during input using the , , , , and keys.

4.4 DOS

Top level menu

The item allows you to access the operating system without unloading *SpecBox*. Type

exit

to return to *SpecBox*.

5 THE SYNTAX CHECKER

The syntax checker parses the input file, constructing a parse tree in the process that is then used by the analyser (Section 6) to locate semantic errors. The syntax checker examines the source file for conformance to the grammar given in Section 9.

5.1 Syntax checker options

The syntax checker allows you to check the syntax of a complete module, or of smaller syntactic units. It also enables you to select the position in the file where checking is to begin. If you wish to return to the start of the file during the course of syntax checking, reselect the file name using the **File** item on the top level menu, or select the built-in editor using the **File** item and press **F9** followed by **F8**.

Syntactic units

Top level menu

Module
tYpe defn
sTate
Value defn
fUction defn
Operation defn
eXpression
Specify

Selecting one of these menu items allows you to check for conformance to the appropriate nonterminal in the grammar,

as follows:

Menu item	Syntactic unit
Module	vdmmodule
Type definition	typedefn
State	stateinfo
Constant definition	valuedefn
Function definition	fundefn
Operation definition	opdefn
Expression	expr

Most other classes can be checked using the Specify menu item. However, since the concrete syntax used by the parser differs from the abstract and printed syntaxes, some nonterminals, particularly at the lower levels of the grammar, are not accessible this way, and an error message is produced if they are specified. You can restrict the parse to these items within a higher syntactic class using the Grammar option on the parser menu (see Section 5.2).

5.2 General operation

The syntax checker reads the input file and assembles the characters into tokens according to the lexical rules given in Section 9. As each token is completed, it is written to the screen, and then checked for grammatical correctness. If the syntax of the input file is completely correct, the message **CHECK OK** will appear in the bottom right of the screen when the end of the syntactic unit has been reached, and the top level menu will be redisplayed. The current line number is displayed at the bottom right of the screen during checking.

If you are checking certain syntactic items, the parser will be unable to deduce if the end of the item has been correctly reached. This always occurs, for example, in the case of an expression, since the following token might be a misspelt binary operator (e.g. `uinion` for `union`). Whenever there is a doubt, the checker displays a menu headed **End of text?**, and highlights the token that it believes is the first one outside the section of text being checked. Answering will successfully conclude the check.

If the file has been successfully rechecked after correction by means of the built-in editor, a menu headed **Save altered file?** will appear before the main menu is displayed; select or according to whether or not you wish to preserve the edits. You cannot analyse the specification or generate a \LaTeX or Mural file from it unless the edits are preserved, since otherwise the outputs would refer to an out-of-date file. If you save the edits, the original file is renamed with the extension `.bak`.

In batch mode, the progress of the syntax checking session is recorded in a log file, named by the source filename with the extension changed to `.log`.

5.3 Selecting the start point

You can use the built-in editor to select a start point for the check other than the start of the file. Choose the item from the top-level menu, position the cursor at the start of the region of text to be checked, and press or click the mouse on the text 'Start point' on the menu bar at the bottom of the screen. Then select or click 'Quit' to return

to the top-level menu. See Section 5.7 for more information on the built-in editor.

5.4 *Selecting an insertion point*

You can pause the syntax checking at any point by placing an *insertion point* in the file. When the check reaches this point, it pauses and the correction menu is displayed, thus allowing you to select possible tokens from the correction menu, using the grammar interrogation feature if required, or insert text using the built-in editor.

An insertion point is placed in the file from the built-in editor by pressing **F7**, or clicking the text **Insert point** with the mouse. A file can contain several insertion points. An insertion point can be removed by deleting it by means of the editor in the same way as any other character. See Section 5.7 for more information on the built-in editor.

SpecBox treats the end of the file in exactly the same way as an insertion point.

5.5 *Action on error*

If a syntax error is encountered, the erroneous token is highlighted, an error banner is displayed, and a *beep* sounds if the system is configured with *beep on*. A diagnostic message such as **Erroneous token: midule** is displayed at the bottom of the screen, and the correction menu is displayed.

SpecBox treats the end of the file as an insertion point rather than an error, and does not display the error message if it is encountered before the end of the check.

5.6 Adding and correcting text

Adding and correcting text is controlled by the correction menu, which is displayed when the checker encounters a syntax error, an insertion point, the end of the file, or if it is not sure whether the correct end of the parse is reached or not. The correction menu is headed ‘Syntax error’ if an error is detected, ‘Insert text’ at an insertion point or the end of the file, or ‘End of text?’ if the end of the parse may have been reached.

The correction menu contains the items , , , and , followed by a list of possible grammar terminals.

The syntax check can be abandoned at this point by pressing or clicking the right mouse button. If you have previously made changes to the input file, another menu will appear headed **Save altered file?**; select or according to whether or not you wish to preserve the edits. If you save the edits, the original file is renamed with the extension **.bak**.

The grammar terminals are generally listed as the terminal strings expected (e.g. `::`, `==` or `<identifier>`), but binary (e.g. `+`, `<>`, `and`, `union`) and unary (e.g. `not`, `hd`) operators are summarised as `<binary operator>` and `<unary operator>`.

Selecting a replacement token inserts it before the highlighted token, or replaces the highlighted token with it, according to the insertion mode selected. If the token is enclosed in angled brackets it is a variable; selecting a variable causes an edit box to be invoked into which the required token should be input. The built-in editor is then invoked, as described in Section 5.7 below.

Edit option

Correction menu..... Edit

Selecting the Edit menu item causes the built-in editor to be invoked, with the cursor at the start of the highlighted token. See Section 5.7 for more information on the built-in editor.

Complete option

Correction menu..... Complete

If the Completion item is available, it means that there exists only one possible token or list of tokens, ignoring optionals, at this point in the parse. Selecting this item will cause these token or tokens to replace or be inserted before the highlighted token, according to the insertion mode selected.

Sometimes, the checker cannot decide whether the end of the parse has been reached. If this is so, it stops with the first token beyond the parsed text highlighted, and the correction

menu headed 'End of text?'. Selecting in this case will cause the check to conclude successfully.

Grammar option

Correction menu

Selecting the item gives menus of the allowable terminals and nonterminals at each possible level in the grammar.

Selecting a nonterminal will cause a menu of terminals and nonterminals one level deeper in the grammar to be displayed. Selecting leads to a menu one level higher in the grammar.

If a token in the source file is highlighted, selecting a terminal will cause it to be inserted before the highlighted token, or to replace the highlighted token, depending on the setting of the insertion mode. Otherwise, the terminal will be placed at the end of the text. If the token is enclosed in angled brackets it is a variable. Selecting a variable causes an edit box to be invoked into which the required token should be input.

The choice of nonterminal is used to restrict the choices offered by the menu item as the parse progresses. The choice provided is intended as an illustration of the basic shape of the grammar at that point, and usually omits some or all of the optional items. The selection offered by the option can of course be overridden by entering any syntactically correct token via the editor, or by returning

to the correction menu and selecting a terminal from there.

As an example, suppose you reach a point in your specification where *expr* is expected. If you select Grammar you will be offered a list of nonterminals that includes *subsequence*. Suppose you select this, enter the editor, type

foo

then reparse. After ‘foo’ is checked, a new correction menu will appear giving all possible terminals, which include:

([~ <binary operator>

However, if you select Grammar again, the only choice presented will be ‘(’, as this is the shortest route to a subsequence. The selection process nests, so that if you selected *subsequence* ... *oldvarname*, after

foo

you would be offered only ‘~’, and after

foo~

you would be offered only ‘(’.

The effect of a choice extends as far to the right as possible, so after

foo~(a,...,b)

you will be offered only ‘(’ again.

Skip option

Correction menu Skip

The Skip item causes the syntax checker to jump forward in the file to the start of the next major syntactic item. A message giving the line number at which checking recommenced is given at the bottom of the screen, or the warning `Cannot resynchronise - parse concluded` if the end of the file is reached before such a place is found. Further analysis is not allowed if syntax errors have been skipped in this way, and the check ends with the message `FINISHED`. This option is available only when checking complete modules.

Insert mode option

Correction menu Insert mode

The Insert mode item toggles between insert mode and overwrite mode for corrections chosen from this menu. The insert or overwrite mode of the editor is not affected.

5.7 Editor

Correction menu Edit

SpecBox's built-in editor is invoked by selecting the option Edit from the correction menu, or by choosing one of the replacement tokens.










In order to conserve memory, the editable text is limited to one screen. If more text is input beyond this limit, the lines that scroll off the top cannot be accessed using the cursor keys. Pressing **F9** causes the file to be reloaded so that the start can be accessed again. This option is disabled during a parse, as otherwise it would be too difficult for the parser to track changes.

The editor can be used to set the start of the check by positioning the cursor at the required position and pressing **F10**. See Section 5.3 for more details.

It can also be used to set an insertion point in the text. Pressing **F7** causes a special character, displayed as a small left arrow, to be inserted in the source. The syntax checker will stop when this character is encountered, and produce the correction menu from which appropriate tokens can be selected. Selecting a token will cause it to be inserted immediately before the insertion point. The insertion point character can be deleted in the same way as any other character when no longer required. See Section 5.4 for more details.

Miscellaneous functions





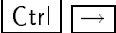
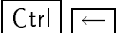



With the keyboard, the following actions control the miscellaneous functions:


Key	Function
	Delete character at cursor
	Toggle insert and overwrite mode
	Delete character to left of cursor
	Mark insertion point
	Quit editor (and reparse if parsing)
	Return to start of file (not available if parsing)
	Create new start of syntax check (not available if parsing)
	Quit (without reparsing)
	Help

With the mouse, editor functions can be selected by clicking on the description in the menu bar at the bottom of the screen. Clicking the right button quits the editor without reparsing.

Cursor movement

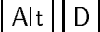

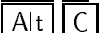
With the keyboard, the following actions position the cursor:



Key	Function
	Move cursor one character to right
	Move cursor one character to left
	Move cursor up one line unless at top of screen
	Move cursor down one line
	Scrolls the view down one line if at bottom of screen
	Move cursor to end of line
	Move cursor to beginning of line
	Move cursor to top left of screen
	Move cursor to bottom left of screen
	Delete character to left of cursor

With the mouse, the cursor can be positioned by pointing to a character and clicking the left button. The view can be scrolled by clicking the left button on the small arrow () at the bottom right of the screen.

Deleting and copying

With the keyboard, the following actions delete and copy text:

Key	Function
	Delete block or line
	Insert block or line
	Copy block or line

When   is pressed, a block, indicated by highlighting in a different colour on a colour screen or reverse video on a monochrome screen, will appear, running from the cursor to the end of the editable area. Moving the cursor to the right of

or below its original position will cause the block to terminate at the cursor; moving the cursor above or to the left of its original position does not affect the block. All the cursor movement keys can be used to position the block as required. The selected area cannot extend beyond the top of the screen, so only the visible highlighted area is affected by the delete operation. Pressing **Enter ↵** causes the highlighted block to be deleted from the text, and stored in the insert buffer. Pressing **Esc** cancels the operation.

Pressing **Alt C** has the same effect as **Alt D**, except that the selected area is not deleted.

Pressing **Alt I** inserts the contents of the insert buffer immediately before the cursor position, and can thus be used for both copying and undeleting text.

With the mouse, text can be deleted by clicking the left button at the start of the area, then clicking 'Delete' in the menu bar at the bottom of the screen. A block, indicated by a different colour on a colour screen or reverse video on a monochrome screen, will appear, running from the cursor to the end of the editable area. Clicking the mouse to the right of or below its original position will cause the block to terminate at that point; clicking above or to the left of its original position does not affect the block. Clicking 'Delete' again causes the highlighted block to be deleted from the text, and stored in the insert buffer. Clicking the right button cancels the operation.

Text is copied in the same way as it is deleted, only 'Copy' is clicked in the menu bar.

Clicking ‘**Insert**’ inserts the contents of the insert buffer immediately before the cursor position, and can thus be used for both copying and undeleting text.

6 THE ANALYSER

Top level menu

Analysers

The *SpecBox* semantic analyser performs basic semantic checking of VDM specifications and is concerned with the *scopes* and *arities* of formulae.

In addition, a number of VDM specific design rules are checked, such as the visibility of state variables in operations specified by pre- and post-conditions.

6.1 Arities

The arity of a value is the number of input parameters that it depends upon, together with the number of results that it can deliver. In a sense, the arity is a simplified kind of *type* and can be checked in a similar way.

In VDM, various items can possess arities in addition to basic functions, such as mappings and operations. There are also families of functions associated with a given VDM specification such as record constructors and selector functions.

6.1.1 Functions

All functions in VDM have functional types:

$$f : (A_1 \times \cdots \times A_n) \rightarrow (R_1 \times \cdots \times R_m)$$

The arity of the function f , with the above type, is (n, m) —that is, it takes n input arguments and returns m results (where $n \geq 0, m \geq 1$).

The arity of so-called curried functions may be determined similarly; the arity then consists of a sequence of two or more input tuples paired up with the number of resulting output values.

6.1.2 Operations

Like functions, VDM operations have an arity which can be derived from the associated type.

$$OP : (A_1 \times \cdots \times A_n) \Rightarrow (R_1 \times \cdots \times R_m)$$

The arity of the operation OP , with the above type, is (n, m) —that is, it takes n input arguments and returns m results (where $n \geq 0, m \geq 1$).

Unlike functions, operations may not be curried.

6.1.3 Records

Associated with each VDM record definition of the form:

$$\begin{array}{lll} RTy & :: & sel_1 : Ty_1 \\ & & \vdots \\ & & sel_n : Ty_n \end{array}$$

is a family of selector functions (which may be overloaded) together with a construction function, $mk\text{-}RTy$, which has

type:

$$mk\text{-}RTy : (Ty_1 \times \cdots Ty_n) \rightarrow RTy$$

Each selector function, sel_i , has type:

$$sel_i : RTy \rightarrow Ty_i$$

6.1.4 Type invariants

A type invariant is a boolean predicate, taking a single argument as input, with type:

$$inv\text{-}Ty : Ty' \rightarrow \mathbf{B}$$

where Ty' is the ‘unconstrained’ type which Ty is a subset of.

6.1.5 State initialisation

The initialisation predicate for a state is a boolean predicate, taking a single argument as input, with type:

$$init\text{-}St : St \rightarrow \mathbf{B}$$

where St is the state space type.

6.2 VDM quotation expressions

There are various places where special VDM functions may be used or quoted. In particular, the pre- and post-conditions of functions and operations may be quoted. This ‘quotation’ makes use of a systematic convention from [1].

6.2.1 Function Quotations

If a function has the following functional type:

$$f : (A_1 \times \cdots \times A_n) \rightarrow R$$

the associated pre- and post-conditions have type:

$$\begin{aligned} pre-f & : A_1 \times \cdots \times A_n \rightarrow \mathbf{B} \\ post-f & : A_1 \times \cdots \times A_n \times R \rightarrow \mathbf{B} \end{aligned}$$

6.2.2 Operation quotations

If an operation (implicit or explicit) has the following operation type, with state space type ST :

$$Op : (A_1 \times \cdots \times A_n) \Longrightarrow (R_1 \times \cdots \times R_m)$$

the associated pre- and post-conditions have type:

$$\begin{aligned} pre-Op & : A_1 \times \cdots \times A_n \times ST \rightarrow \mathbf{B} \\ post-Op & : A_1 \times \cdots \times A_n \times ST \times ST \times R_1 \times \cdots \times R_m \rightarrow \mathbf{B} \end{aligned}$$

In each case, the first occurrence of state type ST represents the state on entry and the second use is the state on exit.

6.3 Scope

All declarations occurring within a specification introduces a scope (i.e. a region of the text) within which the identifier declared can be correctly referred to.

The occurrence of an identifier without a corresponding declaration within the available context is therefore regarded as a scoping error.

Each main class of declaration introduces global names of the following kind:

State types The state declaration introduces the identifier as a record type. This identifier can then be used globally within type expressions. Additional entities are:

- (as for record type declarations)
- the variables declared (as fields) which may be referred to in operations
- state initialisation predicate

Type declarations The type declaration introduces the identifier as a type. This identifier can then be used globally within type expressions. Additional entities are:

- records (e.g. construction function & selector functions)
- quoted literals (defined by explicit enumeration)
- invariant functions

Value declarations The value (or constant) declaration introduces the identifier as a value together with its type (i.e. arity). The identifier can then be used globally within expressions.

Function declarations The value (or constant) declaration introduces the identifier as a value together with its type

(i.e. arity). The identifier can then be used globally within expressions. Additional entities are:

- pre- & post-condition predicates

Operation declarations The operation declaration introduces the identifier as an operation together with its type (i.e. arity). The identifier can then be used globally within statements (i.e. explicit operation calls). Additional entities are:

- pre- & post-condition predicates

6.3.1 Local declarations

The VDM specification language has several ways of introducing local identifiers that have textually limited regions of scope. Moreover, these ways arise syntactically whenever a *pattern* can be used:

- use of state variables in operations
 - formal parameters to operations & functions
 - quantifiers, lambda and iota expressions
 - statement declaration forms (dcl and def) that introduce assignable local variables
 - let and let-be constructs
 - case constructs.
-

6.4 Error messages

During analysis, various errors and problems with the given specification may be encountered. As a result, various messages are generated and sent to the screen and also to the listing file. The messages are as follows:

Arity errors

Such errors arise when the number of arguments to an operation or function call do not correspond to the arity given by declaration.

A.1 Incorrect number of arguments for <name>

The item was declared, but its use is inconsistent with the declaration (e.g. function application, length of tuples).

Incorrect usage errors

Such errors arise in misusing VDM objects in some way. One example of misuse is to use an item that has not been declared. Another is the attempt to use a state-variable in a context in which it has not been imported.

B.1 Undeclared identifier <name> (expecting <class>)

The specified object *name* was not declared, either locally or globally. The context was expecting an object described by *class*.

B.2 Incorrect use of <object> (<expectation>)

This error arises when, for instance, a declared named value is used within a type expression, usage that is inconsistent with its declaration. The *object* indicates what has been found and, when present, the *expectation* indicates what the context required.

**B.3 Item <name> is not declared as a record type
(no mk-constructor function declared)**

A constructor (mk) function has been used for an item that is not a composite object. This arises when, for instance, a constructor function of the form *mk-name* has been used where *name* is not a (known) record type.

**B.4 Non-record type or basic type <name> in
is-expression**

The is-expression *is-name(expr)* has been used where *name* is neither a basic type nor a (known) record type.

B.5 Item <name> not declared as a field selector

An undeclared field called *name* for some composite object has been referenced.

**B.6 Selector <name> used incorrectly as a value
(use dot notation to "apply" selector)**

Fields in composite objects should be selected by the dot notation rather than by using the selector name as a function.

Definition and declaration structure errors

Such errors concern the structure of declarations and definitions and can arise when an inconsistency in this structure has been detected.

An example of this is when an explicit function is said to have two parameters in its type, but its definition then gives three parameters instead.

C.1 Repeated global declaration of <kind> <name>

A redeclaration has been encountered for a globally declared VDM object of type *kind* and called *name*.

C.2 Multiple local decl. of pattern identifier <name>

The named item is declared more than once in the same local declaring context. This error generally arises within patterns in cases and formal parameter lists.

C.3 Local declaration(s) fail to bind any variables

A local declaration (i.e. binding or pattern match) failed to define any variables in a context in which it is required to do so, such as formal parameter lists and quantifiers.

C.4 "MatchValue" pattern occurs in an erroneous context

The 'MatchValue' pattern occurs inside a pattern expression where the overall context forbids its use.

For example, ‘MatchValue’ patterns should not occur anywhere within the parameter lists of functions or operations. On the other hand, they are often used within cases patterns.

A typical cause for such an error is in having too many brackets in pattern expressions.

C.5 Arity mismatch for parameters of <kind> <name>

The (explicit) definition of an object (either function or operation) was given using a different number of arguments to that specified elsewhere in its declaration (i.e. its type).

C.6 Mismatched names in the definition of explicit <kind> <name>

The (explicit) definition of object (either function or operation) was given using conflicting names to that specified elsewhere in its declaration (i.e. its type).

State variable related errors

VDM provides a model-based formalism that makes central use of the notion of *state variable*. Such variables are intended to characterise a system by the way they are used to retain information.

As such, VDM has a number of useful notational conventions concerning the use of state variables in specifications. The errors classified under this section are thus related to the

checking of these conventions.

D.1 Attempted use of <name> as state variable

This error arises when a named item is used as though it were a state variable, when it has not been globally declared as such.

Note that this error can only arise when a global state definition has also been given in the specification.

D.2 State variable <name> has read-only access

The declared state variable *name* was specified to have read-only access. In such cases, explicitly referring to its 'old' value is regarded as an error. This is because such reference implies the potential for changing the value of a read-only variable, which is clearly absurd.

D.3 Global state variable <name> is inaccessible

The declared state variable called *name* was not mentioned as being accessible to the operation, where external clauses have been specified.

D.4 Old value of <name> used incorrectly

This error arises when use is made of the hook notation for the 'old' value of a variable outside a VDM post-condition, such as within the pre-condition to an operation or inside an explicitly defined operation.

Module naming errors

These errors are associated with the names of modules and the items they define.

E.1 Module name <name> clashes with global declaration

In VDM, all objects need to have a corresponding type, including the pre- and post-conditions of operations.

When these are exported from a module the type of the state component is assumed to have the same name as the module (since the actual state type may not have otherwise been exported). This implies that the state type of a module is always externally visible, even though the state type is given a different name internally.

Finally, it means that no other globally declared objects in a module can be given the same name as the module that contains them.

E.2 Terminating module identifier not equal to <name>

The module syntax requires that each module is terminated with the original name of the module that it is finishing.

Warnings

In *SpecBox*, an error is typically something that has to be put right and counts strictly as evidence for a mistake having been

made.

On the other hand, there are occasions where the term ‘error’ would not be strictly correct, but where there is evidence that something is potentially wrong or that it has been misunderstood. In such cases, a warning message is given in place of an error.

W.1 Redeclaring local variable <name> ...
(potential typographical mistake)

A redeclaration of a local variable has been encountered. This may be correct and intentional but it may indicate an error.

W.2 Operation <name> delivers results to statement

At present in draft ISO VDM, operations called directly from a statement should not return results, i.e. they are operations having only pure side-effects. A value-returning operation should be called from appropriate binding statements such as ‘dcl’ or ‘let’ statements.

6.5 Caveats

The notion of arity is an approximation to the general set theoretical notion of type used within VDM. This technique can only check that the number of parameters supplied to a function or operation matches that required by its definition.

As an example of what it cannot yet do is to ensure that the

types of the actual parameters are subtypes of the types of the formal parameters.

6.6 *Output during checking*

During the checking of the specification, the kind of definition encountered is output together with the name of the item that it defines. Error messages are printed as errors are discovered and retained for inclusion in the listing file.

If errors are discovered, a message giving the number of errors is printed at the end of the definition.

6.7 *Report file*

This contains a summary of the modules parsed and of the (global) declarations in each module. These declarations are grouped together for each kind of definition in the VDM language.

Note that the additional VDM specific functions are also included, together with the arity information computed by *SpecBox* for checking purposes. Finally, a statement of the number of errors discovered is given.

File Description: <filename>.rep

6.8 Listing file

This contains a line-numbered listing of the specification modules parsed. Error messages are included near to where the analyser understands that they were encountered.

Naturally enough, any error actually found could possibly be symptomatic of some other, more profound error located elsewhere (usually earlier in the file). In such cases, the user should inspect the context carefully to find the true source of the problem.

As a matter of design policy, the analyser reports each error at most once per phrase and line. Where possible, the analyser attempts to suppress misleading “cascade” errors, which tend to confuse and obscure the possible cause of an error.

File Description: <filename>.lst

6.9 Cross-reference file

This file contains a cross-reference listing of the global identifiers used on a module by module basis. Where possible, declaration sites are also noted.

File Description: <filename>.xrf

7 L^AT_EX GENERATOR

Top level menu Latex

The L^AT_EX generator converts the ASCII source file to L^AT_EX macros that display the specification in draft ISO VDM mathematical syntax. For details of the L^AT_EX document preparation system, refer to [2].

7.1 General operation

The L^AT_EX generator is invoked by selecting Latex. The L^AT_EX output file is named by replacing the extension of the source file by `.tex`.

The generation process only affects that portion of the source file that has been most recently parsed; the remainder of the file is copied unchanged.

The L^AT_EX generator carries out lexical conversion from the ASCII source to L^AT_EX macros that print mathematical symbols as defined in 9.17.

7.2 Formatting

SpecBox uses formatting information in the source file to lay out the L^AT_EX output. In particular:

- It obeys line breaks.

- Single spaces are interpreted as normal interword spaces.
- Two or more spaces tab the output to the corresponding column in the source. This enables items on adjacent lines of the specification to be aligned if required.

Normally, items in the specification will be separated by single spaces, giving the result on the left from the source on the right (□ indicates a space):

<i>character: Pcode</i>	<code>character:□Pcode</code>
<i>key: Key</i>	<code>key:□Key</code>

However, if you wish to align the types in this example, align them in the source making sure there are two or more spaces before each:

<i>character: Pcode</i>	<code>character:□□Pcode</code>
<i>key: Key</i>	<code>key:□□□□□□□□Key</code>

The tab separations are defined by the length `\sbt`, defined in `sb.sty`. This is set up to give good results on a range of specifications, but if you find it is giving too much or too little space in your case, you can reduce or increase it appropriately. (See [2] for details of setting length dimensions.)

7.3 Comments

Both short comments, which run from `--` to the end of the line, and ‘annotations’, which run from the keyword `annotation to end annotation`, are allowed by draft

ISO VDM. The contents of both types of comment are copied unchanged to the \LaTeX file, and you should therefore ensure that comments are acceptable to \LaTeX .

Note that the ‘annotation’ environment will give a ‘wrongly nested environment’ error message if used with \LaTeX versions dated earlier than 24th May 1989. This message is spurious, and the printed specification is not affected.

7.4 Line numbering

Specifications can be printed with line numbers by preceding them by the command `\sbnumberson`. The default on startup is numbering off; this can also be selected with the command `\sbnumbersoff`.

Numbering begins at 1.1 and proceeds at intervals of .1 until a blank line is encountered, when it is incremented to 2.1. This process continues until another `\sbnumberson` command is encountered, when numbering is reset to 1.1.

\LaTeX cross-references can be implemented by placing a `\label` command in a short comment. *SpecBox* adopts a convention that the `--` at the start of a short comment is not printed if the first character of the body of the comment is also a `-`; this is intended to be used for comments containing only non-printing \LaTeX commands. This is illustrated by the following example, which gives the source followed by the mathematical form.

```

Num = nat          -- A type defn
inv n == n < 256   -- -\label{inv}

annotation

The invariant is at line~\ref{inv}.

end annotation

```

```

1.1  Num = N          -- A type defn
.2   inv n △ n < 256

annotation

The invariant is at line 1.2.

end annotation

```

7.5 Subscripts

SpecBox adopts a convention that any part of an identifier following a double underscore is a subscript. Thus `input_0` is printed as *input*₀.

7.6 Running L^AT_EX

The L^AT_EX macros generated by *SpecBox* are defined in a style file **sb.sty**, which is included in the distribution. You should copy this file to the **inputs** subdirectory of your T_EX system. This style needs to be included in the optional arguments to the **\documentstyle** command. Note that version 2.21 of **sb.sty** is not compatible with L^AT_EX produced by *SpecBox* versions before 2.21.

L^AT_EX files require a preamble that is incompatible with draft ISO VDM, and it is therefore normal to use an **\input** or **\include** command to read the file generated by *SpecBox*. Alternatively, you can select a start point after the preamble.

As an example, suppose you wish to write a file, **report.tex** say, to print the specification generated by *SpecBox* from the file **code.vdm**. **report.tex** will need to contain commands similar to the following:

```
\documentstyle[11pt,sb]{article}
...
\begin{document}
...
\input{code}
...
\end{document}
```

You then run L^AT_EX by typing

```
latex report
```

This produces a file **report.dvi**, which is printed by means of

the particular driver for your printer.

7.7 *bsivdm.sty*

sb.sty incorporates extracts from the public domain style file **bsivdm.sty**, in the version dated 30th March 1989, 14:38, written by Mario Wolczko of the Department of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, United Kingdom.

sb.sty is compatible with **bsivdm.sty**; when they are invoked together, *SpecBox* uses the commands `\Forall`, `\Exists` and `\minus` instead of the normal T_EX commands `\forall`, `\exists` and `-`.

8 MURAL TRANSLATOR

Top level menu Mural

The Mural translator enables syntactically and semantically correct specifications to be transmitted to the Mural VDM support tool, so that proof obligations relating to the specification may be generated and discharged.

The specification is transmitted to Mural by Smalltalk code contained in a file named by replacing the extension of the source file by `.mur`.

The user is referred to [3] for a complete description of the Mural system.

8.1 Syntactic variants

SpecBox will translate syntactic variants into the form recognized by Mural. Thus, for example,

$$\exists a: \mathbf{Z}, b: \mathbf{Z} \cdot \dots$$

is translated as

$$\exists a: \mathbf{Z} \cdot \exists b: \mathbf{Z} \cdot \dots$$

8.2 *Unsupported classes*

Mural does not currently support the complete draft ISO VDM specification language. A warning message is produced if any unsupported syntactic classes are encountered, and generally the translated specification will not be acceptable to Mural. The unsupported classes are as follows:

- The following general classes:

```
abortstmt, cases, charlit, defexpr, dont_care,
fundefexplnyd, fundefimplnyd, funtype,
identstmt, idseqpat, interface, isexpr,
istoken, lambda, letbeexpr, letbind, letexpr,
mapcomp, matchval, opdefexplnyd, opdefexpl,
opdefimplnyd, optionaltype, optype, quotetype,
quotlit, ratlit, recordpat, recordtype,
reverse, seq1type, seqlitpat, seqpatidpat,
seqpatid, sequencecomp, setcomp, setinterval,
setpatidval, simplesetpat, stexpr, stmt,
subsequence, textlit, tupleexpr, tuplepat,
typevardecl, typevariable
```

- The following binary expressions:

```
arithmetic divide, arithmetic mod, arithmetic
exponentiation, arithmetic integer division, map iterate,
map composition, function iterate, function composition,
less than or equal, greater than or equal
```

- The following unary expressions:

unary plus, unary minus, arithmetic abs, floor,
distributed set intersection, distributed sequence
concatenation, distributed map merge

- Curried explicit function definitions.
- Pre-conditions in explicit function definitions.
- Exceptions in implicit operation definitions.
- The basic types **Q**, **char** and **token**.

There are also the following restrictions:

- **isnotyetdefn** is supported only for types.
- State initialisation must be written in the form

$$\text{init } \underline{s} \triangleq s = \text{mk-State}(\dots) \dots$$

- Single field selectors only are allowed.

9 THE SPECBOX GRAMMAR

9.1 EBNF concrete syntax notation

The grammar consists of a collection of (context-free) production rules, in a standard way. The EBNF notational conventions used are stated below:

- (1) Each EBNF rule has the form:

non-terminal = rule-body ;

Therefore, each EBNF rule is terminated by a semi-colon symbol.

- (2) Concatenation of items is specified by a comma.
- (3) The bar symbol (|) separates alternative choices within the body of a rule.
- (4) Terminal symbols are quoted in double-quotes (e.g. "abc"). Literal occurrences of a single quote within a terminal string are given literally (e.g. "a'a").
- (5) Non terminal symbols are specified by all lower case identifiers.
- (6) Optional items are specified using the following notation:

0 or 1 occurrences [... contents ...]

- (7) Sequences of items are specified using the following notation:
-

0 or more repetitions { ... contents ... }

The production rules are assumed to operate upon a stream of tokens produced by lexical analysis of a stream of characters; the lexical rules are described in Section 9.17.

An index to the grammar is provided in Section 9.14. The draft ISO VDM keywords are listed in Section 9.15 and the operator precedence is given in Section 9.16.

9.2 Documents

(1) document = vdmmodule, {vdmmodule}
 | defnblock, {defnblock};

9.3 Modules

(2) vdmmodule = “module”, identifier, [“parameters”,
 modsig], [“imports”, importclause,
 {“,”, importclause}],
 [“instantiation”, instclause, {“,”,
 instclause}], [“exports”, modsig],
 [“definitions”, defnblock,
 {defnblock}], “end”, identifier;

(3) importclause = “from”, identifier, modsig;

(4) instclause = identifier, “as”, instance;

(5) instance = identifier, “(”, [substitute, {“,”,
 substitute}], “)”, modsig;

-
- (6) substitute = identifier, “->”, identifier, {“”, identifier};
 - (7) modsig = {modsigitem};
 - (8) modsigitem = typesigmap
| valuesigmap
| funsigmap
| opsigmap;
 - (9) typesigmap = “**types**”, typedescr, {“;”, typedescr};
 - (10) typedescr = identifier, [typedef];
 - (11) valuesigmap = “**values**”, idvaluesig, {“;”, idvaluesig};
 - (12) idvaluesig = identifier, {“,”, identifier}, “:”, type;
 - (13) funsigmap = “**functions**”, idfunsig, {“;”, idfunsig};
 - (14) idfunsig = identifier, {“,”, identifier}, “:”, funtype;
 - (15) opsigmap = “**operations**”, opsig, {“,”, opsig};
 - (16) opsig = identifier, {“,”, identifier}, “:”, optype, [“using”, identifier];
 - (17) defnblock = typedefns
-

-
- | | | |
|--|--|------------|
| | | stateinfo |
| | | valuedefns |
| | | fundefns |
| | | opdefns; |
- (18) typedefns = “**types**”, typedefn, {“;”, typedefn};
- (19) valuedefns = “**values**”, valuedefn, {“;”, valuedefn};
- (20) fundefns = “**functions**”, fundefn, {“;”, fundefn};
- (21) opdefns = “**operations**”, opdefn, {“;”, opdefn};

9.4 Type definitions

- (22) typedefn = identifier, typedef;
- (23) typedef = typedefeqn
| typedefrcd
| isnotyetdefn;
- (24) typedefeqn = “=”, type, [invariant];
- (25) typedefrcd = “::”, field, {field}, [invariant];

9.5 Type expressions

- (26) type = simpletype
-

		uniontype
		tupletype
		mappingtype
		funtype;
(27) uniontype	=	type, “ ”, type;
(28) tupletype	=	type, “*”, type;
(29) mappingtype	=	maptype
		inmaptype;
(30) maptype	=	“map”, type, “to”, type;
(31) inmaptype	=	“inmap”, type, “to”, type;
(32) funtype	=	typety, “->”, type
		typety, “~>”, type;
(33) typety	=	emptytype
		type;
(34) simpletype	=	quotetype
		typename
		basictype
		typevariable
		settype
		seqtype
		optionaltype
		recordtype
		bractype;

-
- (35) `settype` = `“set”`, `“of”`, `type`;
- (36) `seqtype` = `seq0type`
| `seq1type`;
- (37) `seq0type` = `“seq”`, `“of”`, `type`;
- (38) `seq1type` = `“seq1”`, `“of”`, `type`;
- (39) `optionaltype` = `“[”`, `type`, `“]”`;
- (40) `recordtype` = `“compose”`, `identifier`, `“of”`, `field`,
| `{field}`, `“end”`;
- (41) `field` = [`identifier`, `“:”`], `type`;
- (42) `optype` = `typety`, `“==>”`, `typety`;
- (43) `emptytype` = `“(”`, `“)”`;
- (44) `basictype` = `“bool”`
| `“nat”`
| `“nat1”`
| `“int”`
| `“rat”`
| `“real”`
| `“char”`
| `“token”`;
- (45) `quotetype` = `quotlit`;
- (46) `typename` = `identifier`, `{“”`, `identifier`};
-

(47) typevariable = “@”, identifier;

(48) bractype = “(”, type, “)”;

9.6 State definition

(49) stateinfo = “state”, identifier, “of”, field, {field}, [invariant], [“init”, invinitdefn], “end”;

(50) invariant = “inv”, invinitdefn;

(51) invinitdefn = pat, “==”, expr;

(52) isnotyetdefn = “is”, “not”, “yet”, “defined”;

9.7 Constant definitions

(53) valuedefn = identifier, [“:”, type], valuedefncont;

(54) valuedefncont = valuedef
| isnotyetdefn;

(55) valuedef = “=”, expr;

9.8 Function definitions

(56) fundefn = identifier, [typevardecl], fundef;

(57) typevardecl	=	“[”, typevariable, {“,”, typevariable}, “]”;
(58) fundef	=	fundefimpl fundefimplnyd fundefexpl;
(59) fundefimpl	=	pattypeparams, idtypepair, [precond], postcond;
(60) fundefimplnyd	=	pattypeparams, idtypepair, isnotyetdefn;
(61) pattypeparams	=	“(”, [typebind, {“,”, typebind}], “)”;
(62) idtypepair	=	identifier, “:”, type;
(63) precondition	=	“ pre ”, expr;
(64) postcond	=	“ post ”, expr;
(65) fundefexpl	=	“:”, funtype, fundefcont;
(66) fundefcont	=	fundefbody fundefexplnyd;
(67) fundefbody	=	identifier, patparams, {patparams}, “=”, expr, [precond];
(68) fundefexplnyd	=	identifier, patparams, {patparams}, isnotyetdefn;

(69) patparams = “(”, [pat, {“,”, pat}], “)”;

9.9 Operation definitions

(70) opdefn = identifier, opdef;

(71) opdef = opdefimpl
| opdefimplnyd
| opdefexpl;

(72) opdefimpl = pattypeparams, [idtypepair],
[externals], [precond], postcond,
[“errs”, exception, {“,”,
exception}];

(73) opdefimplnyd = pattypeparams, [idtypepair],
[externals], isnotyetdefn;

(74) externals = “ext”, extvar, {extvar};

(75) extvar = readwritemode, identifier, {“,”,
identifier}, [“:”, type];

(76) readwritemode = “rd”
| “wr”;

(77) exception = identifier, “:”, expr, [“->”, expr];

(78) opdefexpl = “:”, optype, opdefcont;

(79) opdefcont = opdefbody
| opdefexplnyd;

-
- (80) `opdefbody` = identifier, patparams, “==”, stmt, [precond];
- (81) `opdefexplnyd` = identifier, patparams, isnotyetdefn;

9.10 Statements

- (82) `stmt` = assignment
| callstmt
| block
| nondetstmt
| forloops
| whileloop
| ifstmt
| casesstmt
| mccarthystmt
| return
| exit
| “skip”
| “error”
| handlerstmt
| letbestmt
| letstmt
| defstmt
| dclstmt;
- (83) `block` = “(”, {dclstmt}, stmt, {“;”, stmt}, “)”;
- (84) `dclstmt` = “dcl”, identifier, “:”, type, [“:=”, exprorcallstmt], “;”;
-

(85) defstmt	=	“def”, equalsdefn, {“;”, equalsdefn}, “in”, stmt;
(86) equalsdefn	=	patbind, “=”, exprorcallstmt;
(87) letbestmt	=	“let”, simplebind, [stexpr], “in”, stmt;
(88) letstmt	=	“let”, letbind, {“,”, letbind}, “in”, stmt;
(89) assignment	=	statedesignator, “:=”, exprorcallstmt;
(90) statedesignator	=	identifier, {“”, identifier}, {designatoritem};
(91) designatoritem	=	designatorarg fieldid;
(92) designatorarg	=	“(”, expr, “)”;
(93) exprorcallstmt	=	expr, [usingstmt];
(94) callstmt	=	identifier, {“”, identifier}, “(”, [expr, {“,”, expr}], “)”, [usingstmt];
(95) usingstmt	=	“using”, statedesignator;
(96) nondetstmt	=	“ ”, “(”, stmt, {“,”, stmt}, “)”;
(97) forloops	=	intloop

		sequenceloop
		setloop;
(98) sequenceloop	=	“for”, patbind, “in”, [“reverse”], expr, “do”, stmt;
(99) setloop	=	“for”, “all”, pat, “in”, “set”, expr, “do”, stmt;
(100) intloop	=	“for”, identifier, “=”, expr, “to”, expr, [“by”, expr], “do”, stmt;
(101) whileloop	=	“while”, expr, “do”, stmt;
(102) mccarthystmt	=	“(”, guardedstmt, {“,”, guardedstmt}, [othersstmt], “)” “mc”, “(”, guardedstmt, {“,”, guardedstmt}, [othersstmt], “)”;
(103) ifstmt	=	“if”, expr, “then”, stmt, {elsifstmt}, “else”, stmt “if”, expr, “then”, stmt, “else”, stmt;
(104) elsifstmt	=	“elsif”, expr, “then”, stmt;
(105) guardedstmt	=	expr, “->”, stmt;
(106) casesstmt	=	“cases”, expr, “:”, casesstmtaltn, {“,”, casesstmtaltn}, [othersstmt], “end”

-
- | | | |
|---------------------|---|---|
| | | “cases”, expr, “:”, casesstmtaltn,
{“,”, casesstmtaltn}, [othersstmt],
“end”; |
| (107) casesstmtaltn | = | pat, {“,”, pat}, “->”, stmt

pat, “->”, stmt; |
| (108) othersstmt | = | “,”, “others”, “->”, stmt; |
| (109) handlerstmt | = | always

trapstmt

rectrapstmt; |
| (110) always | = | “always”, stmt, “in”, block; |
| (111) trapstmt | = | “trap”, patbind, “with”, stmt, “in”,
block; |
| (112) rectrapstmt | = | “tixe”, “{”, trapbind, {“,”,
trapbind}, “}”, “in”, block; |
| (113) trapbind | = | patbind, “ ->”, stmt; |
| (114) return | = | “return”, [expr]; |
| (115) exit | = | “exit”, [expr]; |

9.11 Expressions

- | | | |
|------------|---|---|
| (116) expr | = | applyexpr

subsequence

letexpr |
|------------|---|---|
-

		letbeexpr
		defexpr
		quantifier
		uniqueexpr
		lambda
		isexpr
		recordconstr
		nameexpr
		literal
		bracexpr
		tupleexpr
		setormapexpr
		sequenceexpr
		recordmodifier
		ifexpr
		cases
		unaryexpr
		binexpr
		fieldselect
		polyfuninst
		“undefined”;
(117) defexpr	=	“def”, equalsdefn, {“;”, equalsdefn}, “in”, expr;
(118) letexpr	=	“let”, letbind, {“,”, letbind}, “in”, expr;
(119) letbeexpr	=	“let”, simplebind, [stexpr], “in”, expr;
(120) stexpr	=	“be”, “st”, expr;

-
- (121) `letbind` = `valbind`
 | `letfunbind`;
- (122) `valbind` = `pattypebind`, “=”, `expr`;
- (123) `letfunbind` = `identifier`, [`typevardecl`], `fundef`;
- (124) `unaryexpr` = `unaryop`, `expr`;
- (125) `unaryop` = “_”
 | “+”
 | “floor”
 | “abs”
 | “card”
 | “len”
 | “not”
 | “dunion”
 | “dinter”
 | “power”
 | “conc”
 | “elems”
 | “inds”
 | “tl”
 | “hd”
 | “dom”
 | “rng”
 | “merge”
 | “inverse”;
- (126) `binexpr` = `expr`, `binaryop`, `expr`;
- (127) `binaryop` = “=>”
-

	“and”
	“or”
	“<>”
	“=”
	“<=>”
	“>=”
	“<=”
	“>”
	“<”
	“subset”
	“psubset”
	“in”, “set”
	“not”, “in”, “set”
	“<:”
	“<-:”
	“.:>”
	“:->”
	“^”
	“+”
	“-”
	“union”
	“munion”
	“\”
	“++”
	“*”
	“/”
	“div”
	“rem”
	“inter”
	“comp”
	“mod”
	“**”,

-
- (128) quantifier = existsuniquequant
 | allquant
 | existsquant;
- (129) existsuniquequant = “exists1”, bind, “&”, expr;
- (130) allquant = “forall”, multibind, {“,”,
 multibind}, “&”, expr;
- (131) existsquant = “exists”, multibind, {“,”,
 multibind}, “&”, expr;
- (132) uniqueexpr = “iota”, bind, “&”, expr;
- (133) setormapexpr = emptyset
 | emptymap
 | setenum
 | mapenum
 | setcomp
 | mapcomp
 | setinterval;
- (134) setinterval = “{”, expr, “,”, “...”, “,”, expr, “}”;
- (135) setenum = “{”, expr, {“,”, expr}, “}”;
- (136) emptyset = “{”, “}”;
- (137) setcomp = “{”, expr, “|”, multibind, {“,”,
 multibind}, [comppred], “}”;
- (138) mapenum = “{”, expr, “|->”, expr, {“,”, expr,
 “|->”, expr}, “}”;
-

(139)	emptymap	=	“{”, “ −>”, “}”;
(140)	mapcomp	=	“{”, expr, “ −>”, expr, “ ”, multibind, {“,”, multibind}, [comppred], “}”;
(141)	comppred	=	“&”, expr;
(142)	sequenceexpr	=	sequenceenum sequencecomp;
(143)	sequenceenum	=	“[”, [expr, {“,”, expr}], “]”;
(144)	sequencecomp	=	“[”, expr, “ ”, bind, comppred, “]”;
(145)	recordconstr	=	mkidentifier, “(”, expr, {“,”, expr}, “)”;
(146)	recordmodifier	=	“ mu ”, “(”, expr, “,”, fieldmod, {“,”, fieldmod}, “)”;
(147)	fieldmod	=	identifier, “ −>”, expr;
(148)	lambda	=	“ lambda ”, typebind, {“,”, typebind}, “&”, expr;
(149)	ifexpr	=	“if”, expr, “ then ”, expr, {elseifexpr}, “ else ”, expr;
(150)	elseifexpr	=	“ elseif ”, expr, “ then ”, expr;
(151)	cases	=	“ cases ”, expr, “:”, casesaltn, {“,”, casesaltn}, [othersexpr], “ end ”

-
- | | | |
|-------------------|---|---|
| | | “cases”, expr, “:”, casesaltn, {“,”, casesaltn}, [othersexpr], “end”; |
| (152) casesaltn | = | pat, {“,”, pat}, “->”, expr
 pat, “->”, expr; |
| (153) othersexpr | = | “,”, “others”, “->”, expr; |
| (154) isexpr | = | istoken, “(”, expr, “)”; |
| (155) fieldselect | = | expr, fieldid, {fieldid}; |
| (156) polyfuninst | = | expr, “[”, type, {“,”, type}, “]”; |
| (157) fieldid | = | “.”, identifier; |
| (158) subsequence | = | expr, “(”, expr, “,”, “...”, “,”, expr, “)”; |
| (159) applyexpr | = | expr, [“[”, type, {“,”, type}, “]”,
“(”, [expr, {“,”, expr}], “)”]; |
| (160) nameexpr | = | oldvarname
 identifier, {“~”, identifier}; |
| (161) oldvarname | = | identifier, “~”; |
| (162) braceexpr | = | “(”, expr, “)”; |
| (163) tupleexpr | = | mk-, “(”, expr, {“,”, expr}, “)”; |
-

9.12 Bindings and Patterns

- (164) `bind` = `bracbind`
 | `simplebind`;
- (165) `bracbind` = `“(”, simplebind, “)”`;
- (166) `simplebind` = `typebind`
 | `setbind`;
- (167) `patbind` = `setbind`
 | `pattypebind`;
- (168) `pattypebind` = `pat`
 | `typebind`;
- (169) `typebind` = `pat, “:”, type`;
- (170) `setbind` = `pat, “in”, “set”, expr`;
- (171) `multibind` = `setmultibind`
 | `typemultibind`;
- (172) `setmultibind` = `pat, {“,”, pat}, “in”, “set”, expr`;
- (173) `typemultibind` = `pat, {“,”, pat}, “:”, type`;
- (174) `pat` = `sequencepat`
 | `recordpat`
 | `tuplepat`
 | `patidorval`
 | `setpat`;
-

(175)	setpat	=	simplesetpat setunionpat patidorval;
(176)	simplesetpat	=	“{”, [pat, {“,”, pat}], “}”;
(177)	setunionpat	=	setpat, “union”, setpat;
(178)	sequencepat	=	seqlitpat seqpatid seqpatidpat idseqpat;
(179)	seqpatid	=	seqlitpat, “^”, patident;
(180)	seqpatidpat	=	seqlitpat, “^”, patident, “^”, seqlitpat;
(181)	idseqpat	=	patident, “^”, seqlitpat;
(182)	recordpat	=	mkidentifier, “(”, [pat, {“,”, pat}], “)”;
(183)	seqlitpat	=	“[”, [pat, {“,”, pat}], “]”;
(184)	tuplepat	=	mk-, “(”, pat, {“,”, pat}, “)”;
(185)	patidorval	=	patident matchval;
(186)	patident	=	identifier “-”;

-
- (187) matchval = matchvalue;
- (188) matchvalue = literal
 | “(”, expr, “)”;
- (189) mkidentifier = mktoken;

9.13 Identifiers and basic tokens

- (190) identifier = tokenitem;
- (191) tokenitem = nametoken
 | quotlit
 | mktoken
 | istoken
 | pretoken
 | posttoken
 | invtoken
 | inittoken;
- (192) nametoken = LEXICAL ITEM;
- (193) quotlit = LEXICAL ITEM;
- (194) mktoken = LEXICAL ITEM;
- (195) istoken = LEXICAL ITEM;
- (196) pretoken = LEXICAL ITEM;
- (197) posttoken = LEXICAL ITEM;
-

-
- (198) invtoken = LEXICAL ITEM;
- (199) inittoken = LEXICAL ITEM;
- (200) literal = boollit
| intlit
| ratlit
| charlit
| textlit
| “nil”;
- (201) boollit = “true”
| “false”;
- (202) intlit = LEXICAL ITEM;
- (203) ratlit = LEXICAL ITEM;
- (204) charlit = LEXICAL ITEM;
- (205) textlit = LEXICAL ITEM;

9.14 Index to the grammar

allquant	130	elsifstmt	104
always	110	emptymap	139
applyexpr	159	emptyset	136
assignment	89	emptytype	43
		equalsdefn	86
basictype	44	exception	77
binaryop	127	existsquant	131
bind	164	existsuniquequant	129
binexpr	126	exit	115
block	83	expr	116
boollit	201	exprorcallstmt	93
bracbind	165	externals	74
braceexpr	162	extvar	75
bractype	48		
		field	41
callstmt	94	fieldid	157
cases	151	fieldmod	147
casesaltn	152	fieldselect	155
casesstmt	106	forloops	97
casesstmtaltn	107	fundef	58
charlit	204	fundefbody	67
comppred	141	fundefcont	66
		fundefexpl	65
dclstmt	84	fundefexplnyd	68
defexpr	117	fundefimpl	59
defnblock	17	fundefimplnyd	60
defstmt	85	fundefn	56
designatorarg	92	fundefns	20
designatoritem	91	funsigmap	13
document	1	funtype	32
elsifexpr	150	guardedstmt	105

handlerstmt	109	mapenum	138
identifier	190	mappingtype	29
idfunsig	14	maptype	30
idseqpat	181	matchval	187
idtypepair	62	matchvalue	188
idvaluesig	12	mccarthystmt	102
ifexpr	149	mkidentifier	189
ifstmt	103	mktoken	194
importclause	3	modsig	7
inittoken	199	modsigitem	8
inmaptype	31	multibind	171
instance	5	nameexpr	160
instclause	4	nametoken	192
intlitt	202	nondetstmt	96
intloop	100	oldvarname	161
invariant	50	opdef	71
invinitdefn	51	opdefbody	80
invtoken	198	opdefcont	79
isexpr	154	opdefexpl	78
isnotyetdefn	52	opdefexplnyd	81
istoken	195	opdefimpl	72
lambda	148	opdefimplnyd	73
letbeexpr	119	opdefn	70
letbestmt	87	opdefns	21
letbind	121	opsig	16
letexpr	118	opsigmap	15
letfunbind	123	optionaltype	39
letstmt	88	optype	42
literal	200	othersexpr	153
mapcomp	140	othersstmt	108
		pat	174

patbind	167	sequenceexpr	142
patident	186	sequenceloop	98
patidorval	185	sequencepat	178
patparams	69	setbind	170
pattypebind	168	setcomp	137
pattypeparams	61	setenum	135
polyfuninst	156	setinterval	134
postcond	64	setloop	99
posttoken	197	setmultibind	172
precond	63	setormapexpr	133
pretoken	196	setpat	175
		settype	35
quantifier	128	setunionpat	177
quotetype	45	simplebind	166
quotlit	193	simplesetpat	176
		simpletype	34
ratlit	203	statedesignator	90
readwritemode	76	stateinfo	49
recordconstr	145	stexpr	120
recordmodifier	146	stmt	82
recordpat	182	subsequence	158
recordtype	40	substitute	6
rectrapstmt	112		
return	114	textlit	205
		tokenitem	191
seq0type	37	trapbind	113
seq1type	38	trapstmt	111
seqlitpat	183	tupleexpr	163
seqpatid	179	tuplepat	184
seqpatidpat	180	tupletype	28
seqtype	36	type	26
sequencecomp	144	typebind	169
sequenceenum	143	typedef	23

typedefeqn 24
typedefn 22
typedefns 18
typedefrcl 25
typedescr 10
typemultibind 173
typename 46
typesigmap 9
typety 33
typevardecl 57
typevariable 47

unaryexpr 124
unaryop 125
uniontype 27
uniqueexpr 132
usingstmt 95

valbind 122
valuedef 55
valuedefn 53
valuedfncont 54
valuedefns 19
valuesigmap 11
vdmmodule 2

whileloop 101

9.15 Keywords

The draft ISO VDM keywords are as follows.

Alphanumeric keywords

abs, all, always, and, annotation, as, be, bool, by, card, cases, char, comp, compose, conc, dcl, def, defined, definitions, dinter, div, do, dom, dunion, elems, else, end, error, errs, exists, exists1, exit, exports, ext, false, floor, for, forall, from, functions, hd, if, imports, in, inds, init, inmap, instantiation, int, inter, inv, inverse, iota, is, lambda, len, let, map, merge, mk_, mod, module, mu, munion, nat, nat1, nil, not, of, operations, or, others, parameters, post, power, pre, psubset, rat, rd, real, rem, return, reverse, rng, seq, seq1, set, skip, st, state, subset, then, tixe, tl, to, token, trap, true, types, undefined, union, using, values, while, with, wr, yet.

Symbols

&	()	*	**	+	++	,	-	->
.	...	/	:	:->	::	:=	:>	;	<
<-:	<:	<=	<=>	<>	=	==	==>	=>	>
>=	@	[\]	^	-	'	{	
->		}	~	~>					

9.16 Operator precedence

The operator precedence is as follows.

Binary operators

Prec.	Operator	Prec.	Operator
1	\Rightarrow	5	\triangleright
2	\wedge	5	$($
2	\vee	5	$+$
3	\neq	5	$-$
3	$=$	6	\cup (set union)
3	\Leftrightarrow	6	\cup (map merge)
4	\geq	6	\boxminus
4	\leq	6	\backslash
4	$>$	6	\dagger
4	$<$	7	\times
4	\subseteq	7	$/$
4	\subset	7	div
5	\in	7	rem
5	\notin	7	\cap
5	\triangleleft	7	\circ
5	\triangleleft	8	mod
5	\triangleright	9	\uparrow

Unary operators bind more tightly than any of these.

*Type operators***Precedence Operator**

1	\rightarrow
1	\leadsto
2	$ $
3	\times

9.17 Lexical rules

This section describes the lexical rules obeyed by *SpecBox*, and gives the relationship between the ASCII syntax and the mathematical syntax produced by the L^AT_EX generator.

Identifiers (nametoken)

Identifiers may be composed of upper and lower case letters, digits, underscores (`_`) and primes (`'`). The mathematical form replaces '`_`' by '`-`'. If you wish to use a keyword as an identifier, you must precede it by a `$`, e.g. `$module` or `pre_$types`.

Old state values are represented by a `~` suffix; these are converted to the hooked form in the mathematical syntax. For example, `var~` is converted to \overline{var} .

Greek letters can be represented by preceding the corresponding upper or lower case letter by `#`. Not all the possible upper case Greek symbols (e.g. `#A`) are available in standard L^AT_EX; `sb.sty` substitutes Roman letters for these.

\omicron is printed as *o*.

Ascii	Mathematical	Ascii	Mathematical
#A	A	#a	α
#B	B	#b	β
#G	Γ	#g	γ
#D	Δ	#d	δ
#E	E	#e	ϵ
#Z	Z	#z	ζ
#H	H	#h	η
#Q	Θ	#q	θ
#I	I	#i	ι
#K	K	#k	κ
#L	Λ	#l	λ
#M	M	#m	μ
#N	N	#n	ν
#X	Ξ	#x	ξ
#O	O	#o	<i>o</i>
#P	Π	#p	π
#R	P	#r	ρ
#S	Σ	#s	σ
#T	T	#t	τ
#U	Υ	#u	υ
#F	Φ	#f	ϕ
#C	X	#c	χ
#Y	Ψ	#y	ψ
#W	Ω	#w	ω

Compound identifiers (mktoken etc.)

A compound identifier is formed by prefixing `mk_`, `pre_`, `post_`, `inv_`, `init_` and `is_` to a simple identifier. The \LaTeX generator converts the underscore to a hyphen.

Quoted literals (quotlit)

Distinguished literals are written in upper case letters and placed in pointed brackets, e.g. `<TRIP>`. The mathematical form is TRIP.

Numbers (intlit and ratlit)

Integers are strings of digits that do not contain a decimal point.

Real numbers should be written in the manner `12.3` or `123.4E-5`, which will be converted to `12.3` and `123.4 $\times 10^{-5}$` in the mathematical syntax. At least one digit must occur directly after the decimal point.

Leading zeros are allowed only in the case of the integer `0` and real numbers such as `0.123`.

Character and text literals (charlit and textlit)

Character literals represent single characters and are written in single quotes, e.g. `'x'`. Text literals represent sequences of characters, and are written in double quotes, e.g. `"A string"`.

Single quotes within character literals and double quotes within text literals must be preceded by a `$`; `$` itself is represented by `$$`. Thus `"` is the mathematical syntax corresponding to `'$'`, and `"A string with a $"` corresponds to the source text `"A string with a $$"`.

All other characters appear unchanged in the mathematical syntax.

Comments

Two sorts of comment are allowed by draft ISO VDM. Brief comments run from a pair of dashes to the end of the line:

```
-- This is a comment
```

Longer comments (annotations) are enclosed between the keywords `annotation` and the next `end annotation`.

Draft ISO VDM allows annotations to be distinguished in any convenient way in the mathematical syntax. *SpecBox* places them in an environment named `annotation`, which is defined in `sb.sty` to place the keywords, printed in the keyword font, around the comment. If you wish to distinguish annotations in some other way, you may redefine the annotation environment appropriately; instructions for redefining L^AT_EX environments are given in [2].

Symbols

The following table describes the ASCII representation of the VDM mathematical symbols and the equivalent mathematical form where the two differ significantly.

Ascii	Math	Ascii	Math
x~	\overline{x}	power	\mathcal{F}
&	\cdot	set of t	$t\text{-set}$
*	\times	seq of t	t^*
'	\cdot	seq1 of t	t^+
<=	\leq	inmap a to b	$a \xleftrightarrow{m} b$
>=	\geq	map a to b	$a \xrightarrow{m} b$
<>	\neq	iota	ι
=>	$\overset{\circ}{\rightarrow}$	lambda	λ
->	\mapsto	mu	μ
~>	\rightsquigarrow	bool	B
=>	\Rightarrow	nat	N
<=>	\Leftrightarrow	nat1	\mathbb{N}_1
->	\vdash	int	Z
==	\triangle	rat	Q
**	\uparrow	real	R
++	\dagger	not	\neg
<:	\triangleleft	inter	\cap
:>	\triangleright	union (sets)	\cup
<-:	\triangleleft	merge (maps)	\cup
:->	\triangleright	in set	\in
munion	\mathbb{E}	not in set	\notin
psubset	\subset	comp	\circ
subset	\subset	and	\wedge
^	\cup	or	\vee
dinter	\cap	forall	\forall
dunion	\cup	exists	\exists
inverse	$^{-1}$	exists1	$\exists!$

10 ILLUSTRATIVE EXAMPLES

This section contains some examples of VDM specifications written using the draft ISO VDM grammar. They include a commentary on the differences between this grammar and that used in [1]. The examples are provided in machine-readable form on the distribution disks. Each example is given first in the ASCII syntax and then in the mathematical. Note that these examples may exceed the subset of the draft ISO VDM syntax currently supported by Mural.

For more details on the use of VDM, readers are referred to [1] and [6]. A description of draft ISO VDM module syntax is given in Section 11.

10.1 General observations

Our experience in converting some of our own work into *SpecBox* shows that the use of a tool such as *SpecBox* enforces certain disciplines, such as the complete declaration of variables and the careful use of naming, that are missing from VDM that has not been mechanically checked. It is these issues relating to scoping where we found many errors in some of the VDM we have translated over the past few months. Some of this detail may not be appropriate in a text book or tutorial, where fragments of specifications are used to illustrate a particular point.

There are also stylistic conventions that are useful to adhere to that are not enforced by the language. The use of capitals to denote operations, the use of uppercase as the first letter of

type names, and the following of the general layout of [1], are all practices to be recommended.

10.2 Notes on the ASCII syntax

The raised dot in quantified expressions is denoted by a “&” in the ASCII syntax, e.g. `for all cno1, cno2 in set dom m & cno1 <> cno2`.

Do not use “-” in forming compound identifiers, e.g. `upper-limit`; the underscore “_” is used instead (e.g. `upper_limit`). This generalizes also to quotation of pre/post conditions as in `post_findb`.

Tests for the type of a variable are carried out by an *is-expression*, e.g. `is_real(X)`.

Local variables do need to be introduced (e.g. via quantifiers or as formal parameters to operations etc.). Undeclared variables will be picked up by the semantic analyser.

The specifier is not permitted to define type operators directly in draft ISO VDM. So, for instance, types cannot be of the form `Bag(X)`. However, the parameterised module facility allows one to import parameterised abstract data types to achieve the same effect (see Section 11).

Do not use reserved keywords such as `end` by mistake: it is worth spending a little time learning the keywords. If you can’t remember them while using *SpecBox* they are provided in the *help* facility.

The ASCII representations for the mathematical symbols are mainly given by compound symbols, and are listed in Section 9.17. Again, it is well worth spending some time familiarizing yourself with the ASCII syntax for these symbols.

Do not confuse “==” (*is defined as*) with “=” (*mathematical equality*). Fortunately, the parser is good at spotting when this occurs and recommending the appropriate change.

10.3 Telegram analysis

This example is from [1], page 201.

Notes:

- `in` is a keyword, so is changed to *inpt* here.
 - The position of the binding in the set comprehension is different from [1].
-

```
module Telegram
definitions

types

Input = seq of Telegram;

Telegram = seq of Word
  inv t == t <> [];

Word = seq of char
  inv w ==
    w <> "" and w <> "zzzz";

Output = seq of Report;

Report::  tgm      : Telegram
          count    : int
          ovlen    : bool
```

functions

```
analyse_telegram : Telegram -> Report
analyse_telegram(word1) ==
  mk_Report(word1, charge_words(word1),
    check_words(word1));

charge_words: Telegram -> int
charge_words(word1) ==
  card{j | j in set inds word1 & word1(j) <> "STOP"};

check_words: Telegram -> bool
check_words(word1) ==
  exists w in set elems word1 & len w > 12
```

operations

```
ANALYSE(inpt: Input) out: Output
post
  len out = len inpt and
  forall i in set inds inpt &
    out(i) = analyse_telegram(inpt(i))
```

end Telegram

```

--
-- Telegram analysis example
--

1.1  module Telegram
.2   definitions

2.1  types

3.1  Input = Telegram*;

4.1  Telegram = Word*
.2    inv t  $\triangle$  t  $\neq$  [ ];

5.1  Word = char*
.2    inv w  $\triangle$ 
.3      w  $\neq$  “”  $\wedge$  w  $\neq$  “zzzz”;

6.1  Output = Report*;

7.1  Report::      tgm      : Telegram
.2                  count    : Z
.3                  ovlen    : B

8.1  functions

9.1  analyse-telegram : Telegram  $\rightarrow$  Report
.2    analyse-telegram(wordl)  $\triangle$ 
.3      mk-Report(wordl, charge-words(wordl),
.4        check-words(wordl));

10.1 charge-words: Telegram  $\rightarrow$  Z
.2    charge-words(wordl)  $\triangle$ 
.3      card {j | j  $\in$  inds wordl  $\cdot$  wordl(j)  $\neq$  “STOP”};

```

-
- 11.1 *check-words*: $Telegram \rightarrow \mathbf{B}$
 .2 *check-words*(*wordl*) \triangleq
 .3 $\exists w \in \text{elems } \overline{\text{wordl}} \cdot \text{len } w > 12$
- 12.1 **operations**
- 13.1 *ANALYSE*(*inpt*: *Input*) *out*: *Output*
 .2 **post**
 .3 $\text{len } out = \text{len } inpt \wedge$
 .4 $\forall i \in \text{inds } inpt \cdot$
 .5 $out(i) = \text{analyse-telegram}(inpt(i))$
- 14.1 **end** *Telegram*

10.4 Code

This example is from [1] starting on page 174.

Notes:

- We strictly need **is not yet defined** for *maxs* and *Letter*, and signatures for the functions.
 - The invariant on the type *Mcode* in [1] uses the type *Letter* as though it was an expression—which is picked up by the analyser.
-

```
module Code

definitions

types

Mcode = inmap Letter to Letter
  inv m ==
    forall x: Letter & x in set dom(m);

Pcode = map Letter to Mcode
  inv m ==
    forall x: Letter & x in set dom(m);

Key = map int to Letter
  inv m ==
    exists n: int & dom m = {0,...,n};
```

Letter is not yet defined

```
state
  Code1 of
    c: Pcode
    k: Key
end
```

functions

```
maxs: set of Letter -> Letter
  maxs(l) is not yet defined;
```

```
ptrans: Letter * Letter * Pcode -> Letter
ptrans(kl,ml,code) == (code(kl))(ml)
```

operations

```
CODE(m: seq of Letter) t: seq of Letter
ext rd c: Pcode
  rd k: Key
post len t = len m and
  let l = maxs(dom k) + 1 in
    forall i in set inds t &
      t(i) = ptrans(k(i mod l),m(i),c);
```

```
DECODE(t: seq of Letter) m: seq of Letter
ext rd c: Pcode
  rd k: Key
post len m = len t and
  let l = maxs(dom k) + 1 in
    forall i in set inds t &
```

```
t(i) = ptrans(k(i mod l),m(i),c)
```

```
end Code
```

```

--
-- Code example
--

1.1  module Code

2.1  definitions

3.1  types

4.1  Mcode = Letter  $\xleftrightarrow{m}$  Letter
      .2      inv m  $\triangle$ 
      .3       $\forall x: \text{Letter} \cdot x \in \text{dom}(m)$ ;

5.1  Pcode = Letter  $\xrightarrow{m}$  Mcode
      .2      inv m  $\triangle$ 
      .3       $\forall x: \text{Letter} \cdot x \in \text{dom}(m)$ ;

6.1  Key =  $\mathbf{Z} \xrightarrow{m}$  Letter
      .2      inv m  $\triangle$ 
      .3       $\exists n: \mathbf{Z} \cdot \text{dom } m = \{0, \dots, n\}$ ;

7.1  Letter is not yet defined

8.1  state
      .2      Code1 of
      .3          c: Pcode
      .4          k: Key
      .5      end

9.1  functions

10.1  maxs: Letter-set  $\rightarrow$  Letter
      .2      maxs(l) is not yet defined;

```

11.1 $ptrans: Letter \times Letter \times Pcode \rightarrow Letter$

.2 $ptrans(kl, ml, code) \triangle (code(kl))(ml)$

12.1 **operations**

13.1 $CODE(m: Letter^*) t: Letter^*$

.2 **ext** $rd\ c: Pcode$

.3 $rd\ k: Key$

.4 **post** $len\ t = len\ m \wedge$

.5 $let\ l = maxs(dom\ k) + 1\ in$

.6 $\forall\ i \in inds\ t \cdot$

.7 $t(i) = ptrans(k(i \bmod l), m(i), c);$

14.1 $DECODE(t: Letter^*) m: Letter^*$

.2 **ext** $rd\ c: Pcode$

.3 $rd\ k: Key$

.4 **post** $len\ m = len\ t \wedge$

.5 $let\ l = maxs(dom\ k) + 1\ in$

.6 $\forall\ i \in inds\ t \cdot$

.7 $t(i) = ptrans(k(i \bmod l), m(i), c)$

15.1 **end** $Code$

10.5 Bank Accounts

This example is from [1] starting on page 148. Note that the types *Acno* etc. have been added, and that the dot notation has to be used to reference fields in a composite object (e.g. $(acm(acno)).bal$).

```
module Bank
definitions

types

Acdata::
  own: Cno
  bal: Balance;

Overdraft = nat;

Balance    = int;

Cno        is not yet defined;

Acno       is not yet defined

state Bank of
  acm: map Acno to Acdata
  odm: map Cno to Overdraft

inv mk_Bank(acm,odm) ==
  forall mk_Acdata(cno,bal) in set rng acm &
    cno in set dom odm and bal >= -odm(cno)
```

end

operations

```
NEWC(od: Overdraft) r: Cno
ext wr odm: map Cno to Overdraft
post
  r not in set dom odm~ and
  odm = odm~ union {r |-> od};
```

```
NEWAC(cu: Cno) r: Acno
ext rd odm: map Cno to Overdraft
  wr acm: map Acno to Acddata
pre cu in set dom odm
post
  r not in set dom acm~ and
  acm = acm~ union {r |-> mk_Acddata(cu,0)};
```

```
ACINF(cu:Cno) r: map Acno to Balance
ext rd acm: map Acno to Acddata
post
  r = { acno |-> (acm(acno)).bal
      | acno in set dom acm &
        (acm(acno)).own = cu
      }
```

end Bank

```

--
-- Bank example
--

1.1  module Bank
2    definitions

2.1  types

3.1  Acdata::
4    .2    own:  Cno
5    .3    bal:  Balance;

4.1  Overdraft  =  $\mathbf{N}$ ;

5.1  Balance    =  $\mathbf{Z}$ ;

6.1  Cno        is not yet defined;

7.1  Acno       is not yet defined

8.1  state Bank of
9    .2    acm:  Acno  $\xrightarrow{m}$  Acdata
10   .3    odm:  Cno  $\xrightarrow{m}$  Overdraft

9.1  inv mk-Bank(acm,odm)  $\triangle$ 
10   .2     $\forall$  mk-Acdata(cno, bal)  $\in$  rng acm  $\cdot$ 
11   .3    cno  $\in$  dom odm  $\wedge$  bal  $\geq -$  odm(cno)
12   .4  end

10.1  operations

11.1  NEWC(od: Overdraft) r: Cno
12   .2  ext wr odm: Cno  $\xrightarrow{m}$  Overdraft

```

```

.3  post
.4       $r \notin \text{dom } \overline{odm} \wedge$ 
.5       $odm = \overline{odm} \cup \{r \mapsto od\};$ 

12.1  NEWAC( $cu: Cno$ )  $r: Acno$ 
.2  ext rd  $odm: Cno \xrightarrow{m} Overdraft$ 
.3      wr  $acm: Acno \xrightarrow{m} Acdata$ 
.4  pre  $cu \in \text{dom } odm$ 
.5  post
.6       $r \notin \text{dom } \overline{acm} \wedge$ 
.7       $acm = \overline{acm} \cup \{r \mapsto mk\text{-}Acdata(cu,0)\};$ 

13.1  ACINF( $cu: Cno$ )  $r: Acno \xrightarrow{m} Balance$ 
.2  ext rd  $acm: Acno \xrightarrow{m} Acdata$ 
.3  post
.4       $r = \{ acno \mapsto (acm(acno)).bal$ 
.5           $| acno \in \text{dom } acm \cdot$ 
.6           $(acm(acno)).own = cu$ 
.7           $\}$ 

14.1  end Bank

```

10.6 Binary Trees

This example concerning the reification of binary trees is taken from [1] pages 197 and 254.

Notes:

- draft ISO VDM doesn't allow hooked variables to be defined within patterns, as they are always associated with the old values of state variables.
- *post-FINDB* is of arity 4: input, old state, new state and output.
- an *is-expression* is used in *INSERTB*: *is-Mnode*(\overline{t}).

```

module Tree
definitions

types
Mrep = [Mnode];

Mnode::  lt: Mrep
         mkk: Key
         md: Data
         rt: Mrep
         inv mk_Mnode(lt,mkk,md,rt) ==
           (forall lk in set collkeys(lt) & lk < mkk) and
           (forall rk in set collkeys(rt) & mkk < rk);

Data is not yet defined;

```

Key is not yet defined

state

 State of
 t: Mrep
end

functions

collkeys: Mrep -> set of Key
collkeys(t) ==
 cases t :
 nil -> {},
 mk_Mnode(lt,mkk,md,rt) ->
 collkeys(lt) union {mkk} union collkeys(rt)
 end

operations

FINDB(k: Key) d: Data
ext rd t: Mrep
pre k in set collkeys(t)
post
 let mk_Mnode(lt,mkk,md,rt) = t in
 k = mkk and d = md or
 k < mkk and post_FINDB(k,lt,lt,d) or
 mkk < k and post_FINDB(k,rt,rt,d);

INSERTB(k: Key, d: Data)
ext wr t: Mrep
pre k not in set collkeys(t)

```

post
  (t~ = nil and t = mk_Mnode(nil,k,d,nil)
  or
    (is_Mnode(t~)) and
    let mk_Mnode(lt_old,mkk,md,rt_old) = t~
    in
      (k < mkk and
        exists lt_new : Mrep &
          (post_INSERTB(k,d,lt_old,lt_new) and
            t = mk_Mnode(lt_new,mkk,md,rt_old)
          )
      or
        mkk < k and
        exists rt_new : Mrep &
          (post_INSERTB(k,d,rt_old,rt_new) and
            t = mk_Mnode(lt_old,mkk,md,rt_new)
          )
      )
  )
)
end Tree

```

```

--
-- Tree example
--

1.1  module Tree
.2    definitions

2.1  types
.2    Mrep = [ Mnode ];

3.1  Mnode::  lt: Mrep
.2             mkk: Key
.3             md: Data
.4             rt: Mrep
.5    inv mk-Mnode(lt,mkk,md,rt)  $\triangle$ 
.6    (  $\forall lk \in collkeys(lt) \cdot lk < mkk$  )  $\wedge$ 
.7    (  $\forall rk \in collkeys(rt) \cdot mkk < rk$  );

4.1  Data is not yet defined;

5.1  Key is not yet defined

6.1  state
.2    State of
.3    t: Mrep
.4    end

7.1  functions

8.1  collkeys: Mrep  $\rightarrow$     Key-set
.2  collkeys(t)  $\triangle$ 
.3    cases t :
.4    nil                                 $\rightarrow \{\}$ ,

```

```

.5      mk-Mnode(lt,mkk,md,rt)  →
.6      collkeys(lt) ∪ {mkk} ∪ collkeys(rt)
.7      end

```

9.1 operations

10.1 *FINDB*(*k*: Key) *d*: Data

```

.2  ext rd t: Mrep
.3  pre k ∈ collkeys(t)
.4  post
.5    let mk-Mnode(lt,mkk,md,rt) = t in
.6      k = mkk  ∧ d = md                      ∨
.7      k < mkk  ∧ post-FINDB(k,lt,lt,d)       ∨
.8      mkk < k  ∧ post-FINDB(k,rt,rt,d);

```

11.1 *INSERTB*(*k*: Key, *d*: Data)

```

.2  ext wr t: Mrep
.3  pre k ∉ collkeys(t)
.4  post
.5    (  $\overleftarrow{t}$  = nil ∧ t = mk-Mnode(nil,k,d,nil)
.6      ∨
.7      (is-Mnode( $\overleftarrow{t}$ )) ∧
.8      let mk-Mnode(lt-old,mkk,md,rt-old) =  $\overleftarrow{t}$ 
.9      in
.10      (k < mkk ∧
.11        ∃ lt-new : Mrep ·
.12          (post-INSERTB(k,d,lt-old,lt-new) ∧
.13            t = mk-Mnode(lt-new,mkk,md,rt-old)
.14          )
.15      ∨
.16      mkk < k ∧
.17      ∃ rt-new : Mrep ·

```

```

.18      (post-INSERTB(k,d,rt-old,rt-new) ∧
.19      t = mk-Mnode(lt-old,mkk,md,rt-new)
.20      )
.21      )
.22      )

12.1  end Tree

```

11 MODULES

This section gives some background to the use of the modularization notation in draft ISO VDM. The modularization facility does not form part of the ‘core language’; when the standard is published, the syntax and semantics of the modules will be in an informative annex.

All the examples are syntactically correct, and indeed have been typeset in the mathematical notation using *SpecBox*; however, most give some spurious semantic errors as a result of going outside the currently supported area.

11.1 Background

The module scheme for draft ISO VDM was first proposed by Stephen Bear [4]. The standardization meetings have concentrated on the flat language, although a few papers on modules have been tabled (e.g. [5]). [1] contains some simple examples, one of which is taken up in Section 11.4.1 below.

Modules provide the following facilities:

- A hierarchical design approach.
- Name-space control, by permitting certain objects to be exported. Those not exported are not visible from an external point of view and may only be referenced internally to the module that defines them.
- Reuse of commonly used services.

- Partitioning of services provided—two modules should not provide identical functionality.

The specification of concurrent or distributed systems is explicitly not catered for. The reason is that the semantics of operations assume that the complete system state is known when any state change occurs—this is known as the frame rule. Such an assumption is invalidated when treating concurrent or distributed systems of any realistic complexity. This area is the subject of much recent research by Ketil Stolen and Cliff Jones. Other ISO standard notations, such as LOTOS, are available for formally describing such systems.

11.1.1 Modules as abstract data types

An module may be viewed as something that provides a collection of related facilities and services. These are presented in terms of VDM types and functions, as well as operations over a distinguished type known as the *state type*. Thus a module represents a collection of abstract data types, in terms of types, functions and operations.

It may be the case that a module does not export any operations or a distinguished state type, providing only types and functions externally. Such a module is said to be *pure*; otherwise the module is said to be *state-based*.

A module does not correspond to a task, process or agent. It has no dynamic extent in that modules merely provide services in terms of functions and relations. In short, a module is not “executed” in any operational, mechanistic sense—instead, it just represents a container from which users

of the module can select from the services provided.

11.1.2 Use of modules

In general, a system will consist of a hierarchy of modules, with a single top-level module at the root. Modules in the hierarchy may make use of various facilities provided by other modules, either by instantiating parameterized modules or by importing modules.

All names in draft ISO VDM specifications have a full version of the form

$$Mod`Id$$

where *Mod* is the name of the module where *Id* is defined. (This differs from the proposal in [4], which had a third component for an instantiation.) This form is valid¹ even where *Id* is defined internally to the module. It is only mandatory, however, where it is necessary to disambiguate two items with the same rootname, in the same way as an Ada procedure preceded by:

`use MOD;`

There is no facility for introducing local nicknames of items imported.

Note that the use hierarchy is not recursive and so there is no mutual recursion between items defined in separate modules.

¹ Not currently supported by *SpecBox*.

(Of course, mutual recursion is permitted between items that are defined in the same module.)

The module scheme contains a syntactic class called a *document*, which is composed of a sequence of modules.² The scope of the objects exported by a module is restricted to the document. The top-level module should be the only module not used by any other module contained in the document; it will often be the last one in the list of modules comprising the document.

11.1.3 Importing modules

The simple way of using objects defined in other modules is by *importing* them.

- Importing does not create storage, but provides facilities from the module for defining and manipulating objects.
- As a consequence, a non-parameterized module need never be imported into a module more than once, as it only introduces facilities and services. Importing it twice into the same module can always be replaced by a single import.

² Documents are included in the SpecBox syntax, but not currently accessible through the menu because semantic checking is restricted to single modules.

11.1.4 Parameterized modules

Modules may be parameterized by formal parameters. Such modules are not imported; instead they are *instantiated* by providing actual parameters in place of the formal ones.

- Several uses of a parameterized module may be made within another module, to enable it to be instantiated in a number of differing ways.

11.1.5 General form of the interface

The general pattern for defining and using modules is as follows:

module *ModNm*
interface

parameters

Parameters that this module will need when it is used by modules that instantiate it

ParmModSig

imports

Use of *non-parameterized* standard modules

from *ImpModNm₁*: *ImpModSig₁*
from *ImpModNm₂*: *ImpModSig₂*
etc.

instantiation

Use of *parameterized* modules

```

    InstModNm1 as PModNm1( ParmBind1 )
        ParmImpModSig1
    InstModNm2 as PModNm2( ParmBind2 )
        ParmImpModSig2
    etc.

```

exports

Declaration of publicly visible entities defined
within module *ExpModSig*

definitions

All items declared by this module

end *ModNm*

Note the following:

- If the parameter part is missing, a standard module is defined; the objects it exports are used in other modules by means of an “imports” clause.
- The “imports” clause serves the same purpose as the package specification in Ada. However, an “imports” clause must always be provided where objects from another module as to be used; there is no equivalent of the Ada

with *ImpModNm*;

- In the “imports” clause, the names *ImpModNm* should all be names of standard (i.e. non-parameterized) modules.

- If parameters are given, the module becomes a parameterized module. Parameterized modules are used in other modules by means of an “instantiation” clause, in an analogous way to Ada generic packages.
 - The names of parameters should not clash with other items defined by the module.
 - In the “instantiation” clause, the names *PModNm* must name a specific parameterized module. The binding names in the *ParmBinding* are the names of the formal parameters of the associated module.
 - The modules *ModNm* must be unique within the scope of the module importing them.
 - The names $(InstModNm \cup ImpModNm)$ must be unique.
 - The names *PModNm* need not be unique, to permit several instantiations of the same parameterized module.
 - Module signatures are used to select entities with associated attributes (e.g. type information and class) for introduction into a scope.
 - *ParmModSig*: introduces items for use in “definitions” and the parameter binding part, *ParmBind*. Internally defined items should not be defined using these names. The names declared here are the formal parameter names that are involved in instantiating this module elsewhere.
 - *ImpModSig*: as for *ParmModSig*, so that
-

ImpModNm can be used to disambiguate constructs using name qualification (i.e. *ImpModNm`item*).

- *ParmImpModSig*: introduces items for use in “definitions” part, and the name *ParmInstModNm* may be used to disambiguate constructs using name qualification.
- *ExpModSig*: selects particular items for use in the “exports” part.
- *ParmBind* is a series of bindings of formal parameter names to named entities, each of which may have been provided by a standard module, by a parameter or by a built-in identifier. Note that the present syntax makes it long-winded to use certain built-in operations as module parameters (e.g. try directly providing \geq or even **hd**).
- The hierarchy of module imports and instantiation should be acyclic, where the provision of actual parameters to modules is also taken into account.
- If the “definitions” part is omitted, the exported objects are taken as being **not yet defined**.³

11.2 Sharing

Two types of sharing are of interest: sharing in the sense of mathematical equality, and sharing in the sense of sharing storage.

³ Not currently supported by *SpecBox*.

11.2.1 *Sharing by equality*

Because VDM objects are mathematical abstractions and therefore satisfy the usual properties of equality, it may happen that some of the values (and types) exported by modules may in fact be equal. The sharing rules are as follows:

imported modules These are shared. That is, if objects from a module A are imported into two separate modules X and Y , and then objects from the modules A , X and Y are imported into a fourth module Z , all the exported constructs from A are the same no matter which module they occur in.

parameterized modules In order to be used, parameterized modules must first be instantiated and given a particular name. This name is then used within the instantiating module to refer to the services that the parameterized module exports. If a module is instantiated twice within another module, the constructs must be given distinct names.

In general, parameterized modules will be instantiated differently and will not therefore be equal. However, identical instantiations will be equal; a simple example of this may be constructed by exporting a value that is not dependent upon any of the parameters.

11.2.2 *Sharing of storage*

VDM (in common with Hoare-logic and much else) eschews sharing and insists that parameters are passed by value—it is

normally said that sharing destroys the assignment axiom, but actually it damages much else as well. However, sharing can be modelled by the explicit use of keys, as shown in the following simple example, which defines an abstract data type *STACK* used to set up a collection of stacks in a second module.

```

1.1  module STACK
.2    exports
.3      operations
.4         $PUSH: \mathbf{N} \xrightarrow{o} ()$ ;
.5         $POP: () \xrightarrow{o} \mathbf{N}$ 
.6      types
.7        Stack

2.1  definitions
.2    state Stack of  $s: \mathbf{N}^*$ 
.3    init  $mk\text{-}Stack(s_0) \triangleq s_0 = [ ]$ 
.4    end

3.1  operations
.2     $PUSH: \mathbf{N} \xrightarrow{o} ()$ 
.3     $PUSH(i)$  is not yet defined;

4.1     $POP: () \xrightarrow{o} \mathbf{N}$ 
.2     $POP()$  is not yet defined

5.1  end STACK

6.1  module COLLECTION
.2    imports from STACK
.3      operations
.4         $PUSH: \mathbf{N} \xrightarrow{o} ()$ ;
.5         $POP: () \xrightarrow{o} \mathbf{N}$ 

```

```

.6      types
.7      Stack

7.1     definitions
.2      types
.3      Key is not yet defined

8.1     state Colln of c: Key  $\xrightarrow{m}$  STACK`Stack
.2     init mk-Colln( $c_0$ )  $\triangleq c_0 = \{\}$ 
.3     end

9.1     operations
.2      KPUSH( $k$ : Key,  $i$ : N)
.3      ext wr c: Colln
.4      pre  $k \in \text{dom } c$ 
.5      post  $\text{post-PUSH}(i, \overleftarrow{c}(k), c(k)) \wedge$ 
.6             $\forall k': \text{Key} \cdot$ 
.7             $(k' \in \text{dom } c \wedge k' \neq k) \Rightarrow$ 
.8             $c(k') = \overleftarrow{c}(k')$ 

10.1    end COLLECTION

```

We could have an operation with two keys but notice that

```

KTWO( $k_1$ : Key,  $k_2$ : Key)
ext wr c: Colln
pre  $k_1 \in \text{dom } c \wedge k_2 \in \text{dom } c$ 
post  $\exists i: \text{N} \cdot$ 
       $\text{post-POP}(\overleftarrow{c}(k_1), i, c(k_1)) \wedge$ 
       $\text{post-PUSH}(i, \overleftarrow{c}(k_2), c(k_2)) \wedge$ 
       $\forall k': \text{Key} \cdot$ 
       $(k' \in \text{dom } c \wedge k' \notin \{k_1, k_2\}) \Rightarrow$ 
       $c(k') = \overleftarrow{c}(k')$ 

```

is contradictory in the case where $k_1 = k_2$, which illustrates the potential pitfalls of this type of sharing.

Note that many values of the imported state *Stack* may be used in *COLLECTION* even though the module *STACK* is imported exactly once.

Another example of explicit addressing is shown by contrasting *Mrep* and *Mnode* on page 254 of [1] with *Root*, *Heap* and *Mnode* on pages 255–6.

11.3 Exports and imports

SpecBox currently recognizes simple objects declared in the **imports** part of the interface; however, it does not automatically set up corresponding definitions of pre- and post-conditions and so forth. Thus the following simple example is recognized as correct by SpecBox.

```

1.1  module STUDENTS
.2    imports from NAMES
.3      types Name

2.1    definitions
.2      state
.3        School of sch: NAMES`Name-set
.4      end

3.1    operations
.2      ENROL(n: NAMES`Name)
.3      ext wr sch: NAMES`Name-set
.4      pre n ∉ sch

```

```

.5      post sch =  $\overleftarrow{sch} \cup \{n\}$ 
4.1 end STUDENTS

```

11.4 Instantiation and parameterization

11.4.1 Compiler dictionary

This is taken from pp209–211 of [1]. It addresses the specification of a compiler dictionary. First we define a local dictionary.

```

1.1 module LDICT
.2   parameters
.3     types Id ; Attribs      -- Two parameters defined

2.1   exports
.2     operations
.3       STOREL : Id × Attribs  $\xrightarrow{o}$  ();
.4       ISINL : Id  $\xrightarrow{o}$  B;
.5       LOOKUPL : Id  $\xrightarrow{o}$  Attribs
.6     types
.7       State

3.1   definitions
.2     types
.3       Ldict = Id  $\xrightarrow{m}$  Attribs

4.1   state
.2     State of ld: Ldict
.3     init mk-State(ld0)  $\triangleq$  ld0 = {}
.4     end

```

```

5.1      operations
.2      STOREL (i: Id, a: Attribs)
.3      ext wr ld : Ldict
.4      pre i ∉ dom ld
.5      post ld =  $\overleftarrow{ld}$  ∪ {i ↦ a};

6.1      ISINL (i: Id) r: B
.2      ext rd ld : Ldict
.3      post r ⇔ i ∈ dom ld;

7.1      LOOKUPL (i: Id) r: Attribs
.2      ext rd ld : Ldict
.3      pre i ∈ dom ld
.4      post r = ld(i)

8.1      end LDICT

```

The definition of *LDICT* can now be used in the definition of the main operations, contained in another parameterized module, *CDICT*. Note that according to [4] the state may be implicitly exported and imported, but only if it has the same name as the module, which is not the case here.⁴ We do assume, however, that the pre- and post-conditions of the instantiated operations are imported; we do not need to use their full names when referring to them since they do not clash with any names in *CDICT*.⁵

⁴ The use of the same name for the state and the module would make it clearer that the module was an abstract data type, however.

⁵ However, from the maintenance point of view, it might be better to use the full names for all imported objects.

```

9.1  module CDICT
.2    parameters
.3      types IdC ; AttribsC      -- Two parameters again

10.1  instantiation
.2      ILDICT as LDICT(Id → IdC, Attribs → AttribsC)
.3      types
.4          State
.5      operations
.6          LOOKUPL : IdC  $\xrightarrow{o}$  AttribsC;
.7          STOREL : IdC × AttribsC  $\xrightarrow{o}$  ();
.8          ISINL : IdC  $\xrightarrow{o}$  B

11.1  exports
.2      operations
.3          ENTER : ()  $\xrightarrow{o}$  ();
.4          LEAVE : ()  $\xrightarrow{o}$  ();
.5          STORE : IdC × AttribsC  $\xrightarrow{o}$  ();
.6          ISLOC : IdC  $\xrightarrow{o}$  B;
.7          LOOKUPC : IdC  $\xrightarrow{o}$  AttribsC

12.1  definitions
.2      types
.3          Cdict = ILDICT ` State*      -- Necessary use of
                                           -- name qualification

13.1  state
.2      State of cd: Cdict
.3      init mk-State(cd0)  $\triangleq$  cd0 = [ ]
.4      end

14.1  functions
.2      mins: N-set → N

```

```

.3      mins(s) is not yet defined

15.1    operations
.2      STORE (i: Id, a: Attribs)
.3      ext wr cd : Cdict
.4      pre cd ≠ [ ] ∧ pre-STOREL(i, a, hd cd)
.5      post let cd0 =  $\overleftarrow{cd}$  in
.6          ∃ ld: ILDICT`State ·
.7          post-STOREL(i, a, hd cd0, ld) ∧
.8          cd = [ ld ]  $\frown$  tl cd0;

16.1    ISLOC (i: Id) r: B
.2      ext rd cd : Cdict
.3      pre cd ≠ [ ]
.4      post post-ISINL(i, hd cd, hd cd, r);

17.1    LOOKUPC (i: Id) r: Attribs
.2      ext rd cd : Cdict
.3      pre ∃ j ∈ inds cd · pre-LOOKUPL(i, cd(j))
.4      post let k = mins({pre-LOOKUPL(i, cd(j)) | j: N})
.5          in
.6          post-LOOKUPL(i, cd(k), cd(k), r);

18.1    ENTER ()
.2      ext wr cd : Cdict
.3      post ∃ cd0: Cdict · ILDICT`init-State(cd0)
.4          ∧ cd = [ cd0 ]  $\frown$   $\overleftarrow{cd}$ ;

19.1    LEAVE ()
.2      ext wr cd : Cdict
.3      pre cd ≠ [ ]
.4      post cd = tl  $\overleftarrow{cd}$ 

```

20.1 **end** *CDICT*

11.4.2 Mailing list

This example shows how a sorting module can be instantiated for a particular type and relation in the definition of a mailing list; it is based on [5].

The natural way of writing the instantiation in *mailing-list* would be something like:

$$(item \rightarrow \mathbf{Z}, are\text{-}ordered \rightarrow “\geq”)$$

However, as already mentioned in Section 11.1.5, the grammar requires identifiers for \mathbf{Z} and “ \geq ”. This is achieved below by the local definitions of *Integer* and *ge*; however, a neater solution would be to define another parameterized module giving general expressions for orderings.

```

1.1  module sort
      .2    parameters
      .3    types item
      .4    functions
      .5      are-ordered : item  $\times$  item  $\rightarrow$  B
      .6    exports
      .7    functions
      .8      do-sort : item*  $\rightarrow$  item*

2.1  definitions
      .2    functions
      .3      do-sort : item*  $\rightarrow$  item*
      .4      do-sort(input-list)  $\triangle$ 

```

```

.5      if      input-list = [ ]
.6      then    [ ]
.7      else    insert-sorted(hd input-list, do-sort(tl input-list));

3.1      insert-sorted : item × item* → item*
.2      insert-sorted(elem, list)  $\triangle$ 
.3      if      list = [ ]
.4      then    [elem]  $\frown$  list
.5      else    [hd list]  $\frown$  insert-sorted(elem, tl list)

4.1  end sort

5.1  module mailing-list
.2    instantiation
.3      integer-sort as sort(item → Integer, are-ordered → ge)
.4      functions
.5      do-sort : Integer* → Integer*

6.1  definitions
.2    types
.3      Integer = Z

7.1  functions
.2      ge : Z × Z → B
.3      ge(a, b)  $\triangle$ 
.4      a ≥ b;

8.1      create : Z* → Z*
.2      create(l)  $\triangle$ 
.3      do-sort(l);

9.1      add : Z × Z* → Z*
.2      add(elem, l)  $\triangle$ 

```

```

.3      do-sort([ elem ]  $\curvearrowright$  l);

10.1    delete :  $\mathbf{Z} \times \mathbf{Z}^* \rightarrow \mathbf{Z}^*$ 
.2      delete(elem, l)  $\triangleq$ 
.3      if      l = [ ]  $\overline{\hspace{1cm}}$ 
.4      then    [ ]
.5      else    if      elem = hd l
.6              then    tl l
.7              else    [hd l]  $\curvearrowright$  delete(elem, tl l)

11.1    end mailing-list

```

12 BIBLIOGRAPHY

- [1] C B Jones, *Systematic software development using VDM*, Second Edition, Prentice-Hall International, 1990.
- [2] Leslie Lamport, *L^AT_EX user's guide and reference manual*, Addison Wesley, 1986.
- [3] C B Jones, K D Jones, P A Lindsay and R Moore, *mural: A Formal Development Support System*, Springer Verlag, 1991.
- [4] Stephen Bear, *Structuring for the VDM Specification Language*, in *VDM'88: VDM—The Way Ahead*, R Bloomfield, L Marshall and R Jones (eds), Lecture Notes in Computer Science, no. 328, pp 2–25, Springer-Verlag, 1988.
- [5] G Blaue, *A Copy Rule Approach to the Semantics of BSI/VDM Modules*, BSI IST/5/-/19 no. 216, July 1991.
- [6] John Dawes, *The VDM-SL reference guide*, Pitman, 1991.

13 INDEX

- 80386 version 2.2
 - 8086 version 2.3
 - aborting activities 3.3
 - analyser 1.1, 1.4, 6.1
 - arity checking 6.1
 - error messages 6.7
 - function checking 6.1
 - operation checking 6.2
 - output 6.14
 - quotation checking 6.3
 - scope checking 6.4
 - annotated listing 1.1, 1.4, 6.15
 - audible beep 4.3
 - batch mode 1.5, 3.4
 - bsivdm.sty 7.6
 - config.sys file
 - files command in 2.1
 - configuration 4.2, 4.3
 - cross-reference file 1.1, 1.4, 6.15
 - sorting 4.4
 - default directory 4.1
 - displays 3.3
 - DOS
 - accessing from SpecBox 4.5
 - MS-DOS 5.0 2.2
 - version 2.1
 - edit boxes 3.1
 - editor
 - built-in 1.1, 1.3, 1.4, 5.4, 5.6, 5.9
 - copying 5.12
 - cursor movement 5.11
 - deleting 5.12
 - miscellaneous functions 5.10
 - moving to start of file 5.10
 - setting insertion point 5.10
 - setting start of check 5.10
 - external 1.1, 1.4, 4.2
 - pathname 4.3
 - examples 9.1
 - bank accounts 9.13
 - binary trees 9.17
 - code 9.8
 - telegram analysis 9.4
 - files command
 - in config.sys 2.1
 - garbage collection 3.3
 - grammar 8.1, 9.1
-

-
- access from correction
 - menu 5.7
 - EBNF description
 - 8.1
 - keywords 8.28
 - lexical rules 8.30
 - nonterminals 5.7
 - notes on ASCII syntax
 - 9.2
 - operator precedence
 - 8.29
 - terminals 5.5, 5.7
 - help system 1.2, 3.2
 - highlighting bar 3.1
 - hooked values 8.30
 - installation 2.1
 - LaTeX generator 1.1,
 - 1.4, 7.1
 - handling of comments
 - 7.2
 - line numbering 7.3
 - output file 7.1
 - output format 7.1
 - style file 7.5
 - subscripts 7.4
 - viewing output 7.5
 - lexical rules 5.2, 8.30
 - character literals
 - 8.32
 - comments 8.33
 - identifiers 8.30
 - compound 8.32
 - keywords as 8.30
 - integers 8.32
 - mathematical symbols
 - 8.34
 - old state values 8.30
 - quoted literals 8.32
 - real numbers 8.32
 - text literals 8.32
 - log file 1.1
 - mathematical symbols
 - 8.34
 - menu 3.1
 - cancelling 3.2
 - correction 1.3, 5.4,
 - 5.5
 - complete option
 - 5.6
 - edit option 5.6, 5.9
 - grammar option
 - 5.7
 - insert mode option
 - 5.9
 - skip option 5.9
 - help on 3.2
 - items allowed 3.2
 - scrolling 3.1
 - top level 1.3
 - message
 - analyser error 6.7
 - BATCH MODE 3.5
 - CHECK OK 1.3, 5.2
-

-
- continue option
 - warning 5.9
 - current line number
 - 5.2
 - diagnostic 1.3, 5.4
 - End of text? 5.3
 - FINISHED 5.9
 - line number 5.9
 - Quit? 3.3
 - Save altered file?
 - 5.3, 5.5
 - Microsoft Windows 2.1,
 - 2.2
 - modules 10.1
 - as abstract data types
 - 10.2
 - exports 10.12
 - imports 10.4, 10.12
 - instantiation 10.5,
 - 10.13
 - interface 10.5
 - parameterized 10.5,
 - 10.13
 - sharing 10.8
 - use of 10.3
 - mouse 2.1
 - Mural 1.1, 1.2
 - parse tree 1.1, 1.2
 - path command 2.1
 - quick key 3.1
 - report file 1.1, 1.4, 6.14
 - skip option 1.4
 - source file
 - changed by editor
 - 5.3
 - selecting 1.3, 4.1
 - in batch mode 3.4
 - standard version 2.3
 - starting SpecBox 1.3
 - stylistic conventions 9.1
 - syntactic units 1.3, 5.1
 - syntax checker 1.1, 5.1
 - action on error 5.4
 - adding text 5.5
 - correcting text 5.5
 - insertion point 5.4
 - log file 5.3
 - options 5.1
 - start point 5.3
 - syntax error 1.3
 - virtual drive 4.4
-